

Comparing GNN to traditional Machine Learning algorithms when predicting multiple outcomes in chemotherapy patients.

by Harry Graves

Supervisors: Ivan Olier-Caparroso, Sandra Ortega Martorell

Content

Abstract.....	2
Introduction	3
Mental Health Example.....	4
Survival Analysis	4
Data Source.....	6
Methodology	8
Logistic Regression.....	8
Random Forests.....	9
Dense Neural Network.....	11
Graph Convolution Network GCN	12
Simple Graph Convolution SGC.....	12
Graph Attention Network GAT	13
Pre-processing	14
Results	15
Data Correlation	15
Logistic Regression.....	17
Random Forests.....	20
Dense Neural Network.....	23

Simple Graph Convolution	28
Graph Convolution Network	30
Graph Attention Network	33
Feature Importance.....	36
Conclusions and Further Work	41
Self-Assessment.....	42
References	43

Abstract

In this project, I will be comparing GNN models to traditional machine learning algorithms to predict multiple outcomes in chemotherapy patients, Toxicity and Recurrence. This is a collaboration with Clatterbridge Cancer Centre NHS Trust, they gave access to their data warehouse giving access to all the data they have collected. Clatterbridge have never undertaken a machine learning project before, thus this is filling a gap in research for them and the wider Merseyside region. Using specific localised data, for modelling, should allow these models to be implemented into the patient diagnosis phase of treatment and acquire valuable information on patients and potentially save lives. As GNNs are a newly invented model technology, they have never been implemented on chemotherapy patients, as a result I have compared it to an application in mental health prediction, survival analysis and patient's health records.

The bespoke dataset, I created, had demographic and lab test variables, in the form of numerical data types, comorbidities in binary data type and finally outcomes in binary form of the two labels, toxicity and recurrence. All linked together through the PASID, a unique patient identifier.

I implemented 6 different machine learning Algorithms, Logistic Regression, Random Forests, Dense Neural Network, Simple Graph Convolution, Graph Convolution Network and Graph Attention Transformers. The best model was found to be Dense Neural Network with 70% AUC score over the 2 labels. When comparing to the literature, we would expect much higher AUC scores, I believe the issue is in the data collection phase, as my models were run on only 5490 patients. To improve further on this score, I recommend extracting more samples of each label, rerunning the models with only the important features and trying different types of GNN technology.

Introduction

Cancer represents a formidable global health challenge, affecting individuals across diverse demographics with alarming frequency. Epidemiological data from 2015–2017 illustrate the stark prevalence of the condition, with an estimated 39.5% of both men and women projected to confront a cancer diagnosis at some stage in their lives (www.cancer.gov. (2015)). Particularly distressing is the prevalence of cancer among the vulnerable demographic of children and adolescents, with projections for 2020 anticipating 16,850 new cases within this cohort, tragically culminating in 1,730 fatalities. Within the scope of cancer treatment, chemotherapy assumes a pivotal role, administered to an extensive 39% of afflicted individuals. However, the therapeutic promise of chemotherapy is juxtaposed with its inherent toll on the human body, with adverse effects manifesting in an overwhelming 97.4% of cases (Katta, B. et al (2023)). This study aims to understand a diverse range of adverse effects caused by chemotherapy, such as Toxicity and recurrence. Due to the severe extent of these adverse effects, this study should highlight the importance of early detection and prevention when trying to protect improve the wellbeing of patients.

This project will be completed in collaboration with Clatterbridge Cancer Centre NHS Trust. They have given me access to their whole data warehouse; I will proceed by extracting the relevant patient data and chemotherapy outcomes to use as features and outcomes to model risk levels of certain patients experiencing said adverse effects. Due to the large amount of data collected by Clatterbridge, routine methods of data collection and manipulation will not be functional for this task. Instead, the data will be accessed through the cloud and specific queries can be used to grab relevant data. These queried subsets can be downloaded in CSV format and uploaded into python much like any normal dataset. Clatterbridge have never tried to complete a machine learning project on their data before, thus this is the first of its kind, both for Clatterbridge and for the Merseyside region. Using local data to model patients will increase the reliability of the model when predicting other patients from Merseyside, this can improve the healthcare outcomes for people and improve economic performance and efficiency in the region.

To complete the modelling for this task I will perform a variety of machine learning algorithms, ranging from very simple models such as logistic regression up to and including complex Dense Neural Networks. We will then compare these results to the newly invented Convolution Graph Neural Networks (GNN) (Zhang et al (2019)). Improvements in this field are essential to enhancing personalised medicine and optimising treatment strategies. Due to the unique structure of non-Euclidian graphs (ieee.org. (2024)), these models allow clinicians to tailor treatment plans to individual patients; by predicting their likelihood of experiencing said adverse effects, toxicity and recurrence. By representing the data in graph form it allows us to potentially find hidden correlations between adverse effects and the different demographic, biological features (represented as features of nodes) and comorbidities (represented by edges), resulting in a unique way of identifying potential risk factors in patients.

For completeness, I will implement 3 versions of the GNN model, the standard convolution graph neural network (GCN) (Zhang et al (2019)), Simple Graph Convolution (SGC) (Wu et al (n.d.)) and Graph Attention Transformer (GAT) (Veličković et al (n.d.)) and compare the results

through metrics such as ROC AUC score and computational efficiency. These metrics will evaluate both the performance of the model, and how efficiently the model takes to run, this is a balance where the user must find parity, as increasing the performance, will in most cases make the model less efficient and more time consuming. From a theoretical understanding of the models, we can expect the GCN model to perform the worst of all the GNN models, with SGC having similar results, but using less computational cost and GAT should have the best modelling performance but take significantly more time and energy to perform.

When the best model has been developed, we can use a feature extraction algorithm to find the variables that effect the decision boundary between the prediction of the different adverse effects. These important features will imply some sort of causal relationship between adverse effects and demographic variables and can help identify which specific patient characteristics is shown to increase the risk of developing certain adverse effects.

Mental Health Example

GNN models have not yet been implemented on chemotherapy data in the way I intend to. However, comparable machine learning pipelining have been implemented in similar medical fields. For example, in 2022, 3 GNN models were compared to 2 baseline models, using mental health data (Lu et al (2022)). To complete the analysis a bipartite graph was constructed from the patient and disease data. The nodes represent the patients and contain features that describe the demographic and lifestyle characteristics, such as age, gender, smoking. These nodes were connected through edges representing conditions common to both patients such as depression, alcoholism and psychosis. This Bipartite graph is then converted into a unipartite graph that captures the relationships by the bipartite graph projection. If there is a latent relationship between 2 patients (they have the disease) there will be a connection between them.

Methods	Accuracy	Precision	Recall	F1-score	AUC
Node2Vec	0.7549	0.7654	0.7549	0.7518	0.7396
WYS	0.8259	0.8268	0.8259	0.8257	0.8254
GCN	0.8561	0.8580	0.8561	0.8558	0.8555
GraphSAGE	0.8828	0.8896	0.8828	0.8653	0.8817
GAT	0.9059	0.9073	0.9059	0.9057	0.9054

Figure 1

As you can see, when using the convolution neural networks on this graph structure, a significant increase in ROC AUC was recorded. The baseline models of Node2Vec and WYS scored 0.7396 and 0.8254 respectively, whereas the GCN, GraphSAGE and GAT scored 0.8555, 0.8817 and 0.9054 respectively, showing an average increase in ROC AUC score of 0.0984 or 9.8%. This implies that we can expect an improvement over the baseline machine learning algorithms with the graphical neural networks.

Survival Analysis

In 2023, Graph Neural Networks were also used to calculate survival analysis, by combining cox proportional hazards and deep neural networks, into a GNN-surv model (So Yeon Kim (2023)). This model is a state-of-the-art survival method to model interplay between patients' covariates and treatment effectiveness to facilitate personalised treatment recommendations. This is

suggesting that we should be able to use our GNNs in a similar way and give our chemotherapy patients personalised treatment as well.

The paper proposed 4 different variations on the GNN architecture, 3 of which will be common in my method:

Model	Logistic Hazard ($\mu = 0.2$)		PMF ($\mu = 0.8$)	
	C^{td}	IBS	C^{td}	IBS
MLP-surv	0.5543 \pm 0.0689	0.3183 \pm 0.0497	0.5941 \pm 0.0629	0.2324 \pm 0.0222
GCN-surv	0.6309 \pm 0.0481	0.2331 \pm 0.0358	0.6265 \pm 0.0493	0.2130 \pm 0.0231
SAGE-surv	0.6247 \pm 0.0505	0.2331 \pm 0.0389	0.6378 \pm 0.0415	0.2140 \pm 0.0238
GAT-surv	0.6352 \pm 0.0520	0.2341 \pm 0.0365	0.6339 \pm 0.0451	0.2154 \pm 0.0229

Figure 2

As you can see, they have used two different scoring systems, Logistic Hazard score, the standard scoring system for cox regression and PMF, probability mass function. In the first instance, the best model was GAT, with a score of 0.6352, whereas when comparing PMF, the best model was SAGE. As previously explained, we would expect GAT to be the best scoring model, thus this paper is suggesting we give more validity to the GraphSAGE model as well.

Patient diagnosis example

In 2022, a paper was written (Diaz Ochoa, J.G. and Mustafa, F.E. (2022)) predicting patient procedures using the electronic health records and diagnoses they hold. Again, they compare the results of the baseline GNN model (GCN) to the traditional machine learning algorithms of Random Forests, Logistic Regression Extra Trees, K Nearest Neighbour and Decision Trees. We can see that in all three metrics of recall precision and f1 score the GNN had an improvement in accuracy.

	RF	LR	ET	KNN	DT	GNN Model	Δ
R (in %)	54.0	62.9	58.5	49.2	64.1	62.0	4.26
P (in %)	88.7	76.5	91.8	87.1	61.2	87.0	5.94
$f1$ (in %)	67.2	69.0	71.4	62.9	62.6	73.1	6.48

Overall, from looking at other papers in this field, there is good evidence to suggest that GNN modelling should give an increase in accuracy when being applied to the new domain of

chemotherapy data. Despite this never being attempted before, on either Clatterbridge data or Chemotherapy domains, I believe that the modelling should be successful and the GNN models should give the largest AUC score. My project is a culmination of these 3 papers, comparing traditional machine learning to multiple graph neural networks, in multiple outcome labels.

Data Source

For this project, I had access to all the data stored by Clatterbridge Cancer Centre, an NHS trust based in Wirral. Due to the large amount of data stored, we had remote access to their whole server and would access specific tables using SSMS (SQL Server Management Studio). Using this data, we could construct CSV files to then be used in our modelling.

X input data will constitute the demographic information, lab tests and comorbidities. This will be what the model uses to train to try to predict the adverse effects in the patients. These were all stored in one table with float data types, all linked to individual patients through a unique identifier PASID, which would then be used to link the patient information to the adverse effect outcomes of the chemotherapy. Overall, this gave 67 columns.

The patient demographics used in the final dataset build were 'Age', 'BMI', and 'Sex'. Age was given in years and sex was given as a string of either M for male or F for female, we later converted these categorical features into floats, 1 being male 0 being female to keep it consistent with the other fields. BMI on the other hand was recorded for all the patients and converted into 5 categories from 0 to 4, 0 representing Underweight, 1 representing Healthy, 2 overweight, 3 class 1 obese and 4 class 3 obese. The BMI was recorded by nurses in hand-written notes, and held in a table called `ONC_NurseAssessments`, as a result some values were not given in the correct format or had spelling mistakes in them, thus these patients had to be removed, reducing the amount of patients used in the final training data, however enough correct examples were found. Age and Sex did not have this problem as they were recorded for all chemotherapy patients, in a table called `SACT_CET`. In previous iterations more demographic fields were used such as Religion, Town and ethnic origin, however some needed to be removed. An emptiness threshold was applied where fields with 50% or more NaNs were removed for completeness, as imputing a large percentage of values in a column will reduce the accuracy of values in the column and could skew the model. Other fields had to be removed as they had too many different categories, for example Religion has 19 different unique categories, this is too many as too few examples of each group were given in the data.

Another part of the input data was different medical conditions or comorbidities patients could have, the fields they include are:

Smoking_status	diabetes	hypertension	COPD	hepatitis	HIV
Learning_D	Mental_h	Acute Oncology	Skin & Melanoma	Specialist & Sarcoma	Urology
CNS	Head & Neck	Gynaecology	Lymphoma	Lower GI	Lung
Breast	PASID				

These columns were stored in a binary form, where 1 represents a positive case and 0 a

negative case, again this information had to be taken from the hand-written notes in `ONC_NurseAssessments` table. They include both separate conditions not usually connected with cancer such as diabetes, learning difficulties and HIV, and common types of cancer the patient might have such as Lung Breast or reproductive organs. The rest of the columns represent different lab tests performed on the patients, in the form of float values.

PASID	ACA	ALT	APTT	AST	Adjusted Calciu
Albumin	Alkaline Phosph	Anion Gap	Basophil count	bicarbonate	Bilirutin TII
CRP	Calcium	Chloride	Creatinine	Eosinophil cou	Eosinophil coun
GGT	Globulin	Glucose	HCT	HGB	Haematocritl ev
Haemoglobin	Inorganic Phosp	K	Lymphocyte coun	MCH	MCHC
MCV	Magnesium	Monocytes	NEUT	Neutrophil	Nucleated Rbcs
PLT	Phosphate	Platelet count	Potassium	Prothrombin Tim	RBC
RDW	Serum Creatinin	Sodium	Total Bilirubin	Total Protein	Urea
WBC	WCC	eGFR	Age	BMI	Sex

The lab tests include levels of different elements such as magnesium, sodium and potassium, and counts of important blood features such as platelet, eosinophil and Lymphocyte. These lab tests are performed when patients are first admitted, before any treatment has been administered. As each patient had to have either a nurse assessment or lab test performed individually, some empty values were present, for modelling, the input dataset must be complete, any missing values were imputed using the mean of the column, the mean was selected as it allows any synthetic data to have less of an effect of the prediction and should keep the accuracy of the model as high as it can be. Again, some lab test fields had to be removed as they had more empty values than the threshold of 50% for the selected chemotherapy patients. The lab test results were taken from the `Ds.lab_tests` table.

The adverse effects were used as Y data telling the model that this data is the output of the model. The adverse dataset in a binary dataset, having a unique identifier in a PASID then having a column for Toxicity and a column for Recurrence. These are binary columns where 1 represents a positive case of the adverse effect and 0 represents a negative case. This data was grabbed from the 'SACT_Summary' table, which is part of a group of tables called SACT, holding all the information on different treatments given to patients. Specifically, the `SACT_Summary` table had columns for treatment type and treatment called `DrugTreatmentType` and `Treatment_outcomes` respectively. I could then filter these columns to

only display 'CHEMO' for the treatment type and either 'Toxicity' or Recurrence in either case. Other treatment types included in this table were Bisphosphonate, Hormone therapy, Immunotherapy, Tyrosine Kinase inhibitors, NULL and Other. Other treatment outcomes included, completed as planned, Patient choice, Death, NULL and Other. From this filtration we gained this dataset:

	PASID	Treatment_outcome		PASID	Treatment_outcome
1	001588	Progressive/recurrent can	1	2105152	Toxicity
2	003477	Progressive/recurrent can	2	2105182	Toxicity
3	004609	Progressive/recurrent can	3	2104970	Toxicity
4	005685	Progressive/recurrent can	4	1409938	Toxicity
5	008320	Progressive/recurrent can	5	2105223	Toxicity

Figure 3

This Treatment outcomes column could then be combined and converted, using the python package Pandas, into the binary structure described previously. We also took several 'completed as planned patients', to have a cohort of patients where no adverse effects were counted. Overall, we had 2120 cases of toxicity, 1805 cases of recurrence and 1565 cases where the treatment was completed without adverse effect.

As the GNNs use non-Euclidean graphical data, different structure of input data was required than when used in more traditional machine learning algorithms. Instead of having 1 input dataset, the data was split into two separate datasets, the demographic information and lab tests were in one dataset would populate the nodes information and the binary conditions would be saved in a separate dataset and would be used to calculate the edges connecting the patient nodes. This will create neighbourhoods of nodes with similar conditions, allowing the model to understand the relationship between patients to a deep level and allow for convolution in the neighbourhoods. The same threshold and imputation preprocessing steps were performed on both datasets and the Y data is in the same structure in both types of models.

Methodology

Using this data, we will compare various machine learning models ranging in complexity, evaluating their accuracy through metrics such as ROC AUC and computational efficiency.

Logistic Regression

The first model I will use is logistic regression, a simple and computationally efficient machine learning algorithm. It is often a baseline to determine if a more complex model is necessary. Unlike linear regression, which outputs continuous values, logistic regression produces a binary outcome (1 or 0), representing a true or false case of an adverse effect. Instead of fitting a straight line, it uses an 'S' shaped logistic function.

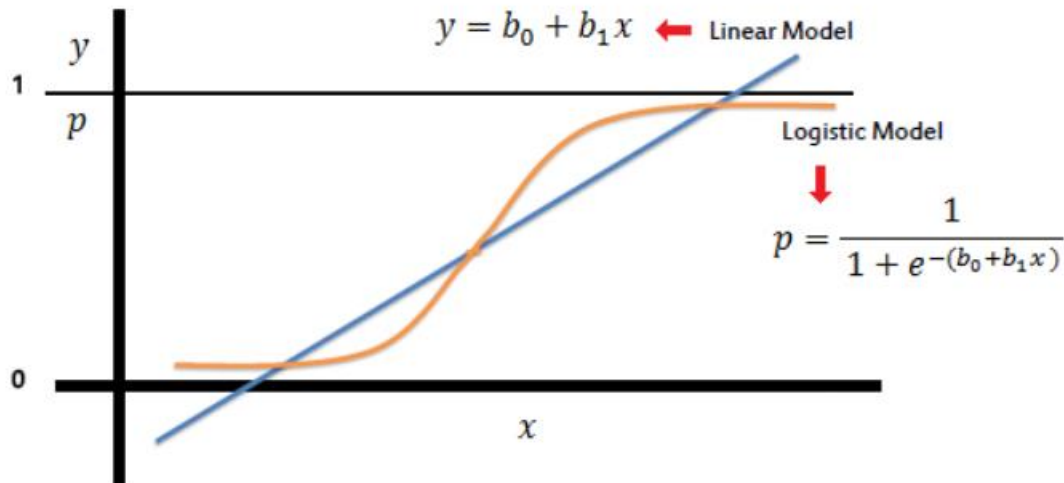


Figure 4

(Saedsayad.com. (2019))

The curve ranges from 0 to 1, representing the probability of a positive or negative outcome. A threshold of 0.5 is applied: probabilities above 0.5 result in a positive result of 1, while those below result in 0. This functions equation is:

Equation 1

Logistic Model

$$p = \frac{1}{1 + e^{-(b_0 + b_1 x)}}$$

where the beta parameters are the input values given to the model.

Random Forests

Random Forests, a more complex ensemble algorithm, improves on Decision Trees by creating multiple trees through bootstrapping. Each tree splits the dataset into cohorts based on decision nodes, ultimately leading to a prediction class. Individual decision trees are sensitive to input data and have high variance, making them less generalisable. Random Forests address this by

training multiple trees on random subsets of the data and using majority voting to determine the final classification, reducing variance and improving generalisation.

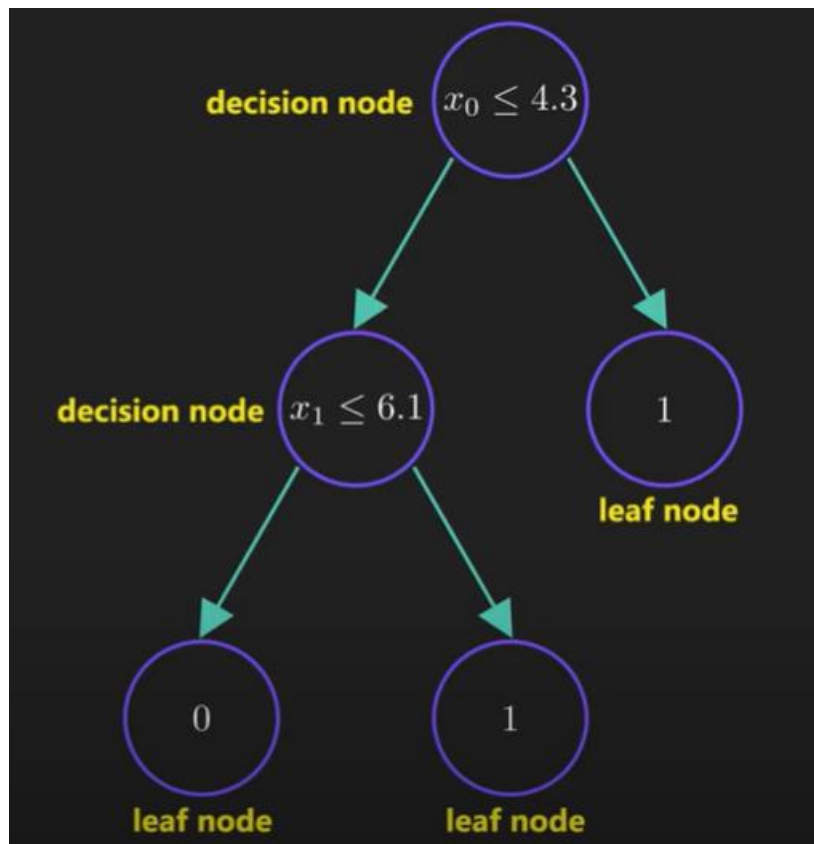


Figure 5

(Hastie et al (2001))

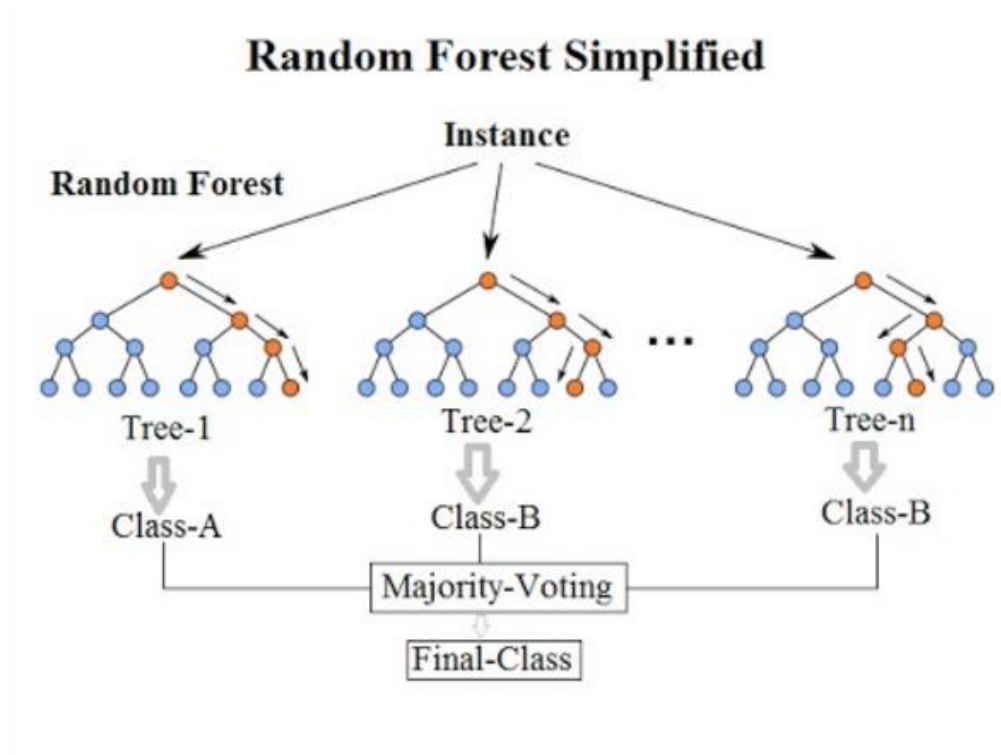


Figure 6

(Koehrsen, W. (2020))

Dense Neural Network

The Dense Neural Network is a neural network with multiple layers between the input and output layers, enabling it to model complex, non-linear relationships. Composed of neurons, each neuron processes inputs through a weighted sum, and a non-linear activation function, passing the output onto the next layer. The network includes an input layer, multiple hidden layers, and an output layer. Hidden layers, not directly exposed to the input or output, allow the DNN to learn increasingly abstract patterns as data moves through the network.

The process of passing data through the network is called forward propagation, where data flows through the input layer, through the hidden layers, to the output layer. Each neuron applies a non-linear activation function, such as ReLU, sigmoid or tanh, to determine if it should pass its signal forward. ReLU outputs the input directly if positive, or zero if negative. Sigmoid compresses outputs between 0 and 1, while tanh compresses them between -1 and 1 . In my case, I chose used ReLU.

After forward propagation, the network accuracy is assessed using a loss function that compares the output to the actual target, measuring prediction error. Common loss functions include Mean Squared Error (MSE) for regression and Cross-Entropy for classification. For our Multi-label classification task I used Binary Cross Entropy with logit loss (BCEwithLogitLoss in PyTorch) for better numerical stability, performance and gradient optimisation. This combines the BCELoss and sigmoid function into a single, more stable operation, reducing the risk of vanishing gradients when outputs are near 0 or 1.

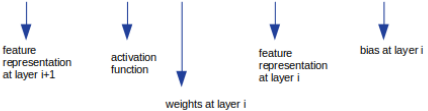
Finally, the DNN uses backpropagation to update neurons weights based on the loss function, improving predictions. Backpropagation uses gradient descent, adjusting the weights in the direction that reduces error. A positive gradient indicates the prediction should be reduced, while a negative gradient suggests it should be increased. If the loss function is large, overfitting might be occurring. To counteract this, dropout layers can be added to disable neurons during training, or regularisation can be applied to penalise large weights.

Graph Convolution Network GCN

Being compared to these models are 3 versions of the GNN model, as this model using a non-Euclidean structure, the input data had to be transformed but the values were all kept the same. I used the standard Graph Convolution Network GCN, the Simple Graph Convolution SGC and Graph Attention transformers GAT.

GCN is built upon the already successful CNN image recognition Neural Networks, just applied to an unstructured, undirected graph. It uses a message propagation method where nodes send their features across edges into neighbouring nodes. It will then apply an aggregation function over the neighbourhood leading to feature smoothing, so nodes can carry information about their neighbours. Then, pass this aggregated vector through a Dense Neural Network, outputting a new vector representation of the node. This process is then repeated for each node to understand relationships within the graph. This method is scalable as many layers can be added using the output of the previous Dense layer as an input, however, can be computationally expensive when compared to the more modern algorithms. The equation given for this algorithm is:

Equation 2

$$H^{[i+1]} = \sigma(W^{[i]} H^{[i]} + b^{[i]})$$


(Mayachita, I. (2020))

The algorithms equation includes both weight matrix and a bias vector. The weight matrix, learned through gradient descent, assigns importance to features affecting the decision boundary-the higher the weight, the more influence the corresponding input has. The bias vector is an additional parameter added to the linear transformation, allowing the model to shift the activation functions input. This flexibility helps the model learn complex patterns by enabling neurons to adjust their outputs independently of their inputs, allowing decision boundaries that do not have to pass through the origin.

Simple Graph Convolution SGC

Simple Graph Convolution (SGC) is a streamlined version of the traditional Graph Convolutional network that aims to reduce the computational complexity while maintaining performance. SGC simplifies the model by removing the non-linear activation functions and collapsing multiple convolution layers into a single linear transformation. These simplifications should not negatively impact performance but reduces computational cost by dramatically speeding up training time,

allowing it to have strong scalability, working best on large datasets. The equation is given as such:

Equation 3

$$H^{(l+1)} = \sigma(\hat{A}H^{(l)}W^{(l)})$$

As you can see the activation function is applied at the end of the algorithm instead of at each layer, alternatively the multi-layer aggregation into a single linear transformation in the form:

Equation 4

$$Z = \hat{A}^K XW$$

Where Z is the output feature matrix, \hat{A}^K is the normalized adjacency matrix raised to the power of K, representing K steps of neighbourhood aggregation, X is the input matrix and W is the weight matrix.

The linear transformation removes the need for multiple matrix multiplications, thus the same operation can be accomplished through a single power operation, reducing the number of individual calculations and reducing computation time.

Graph Attention Network GAT

The final model implemented is the Graph Attention Network (GAT), a major advancement in GNN technology. GAT addresses the challenge of capturing complex graph relationships by introducing attention mechanisms that assign different weights to neighbouring nodes based on their importance. This allows each node to selectively aggregate information from the most relevant neighbouring nodes based on their importance. GAT optimises these attention weights through gradient descent, enhancing its ability to capture intricate graph patterns. Its flexibility, scalability and efficiency in processing large-scale graphs makes it one of the best GNN models for tasks requiring fine-grained node interactions and long-range dependencies.

The equation for the attention heads is given as:

Equation 5

$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \quad (1)$$

$$e_{ij}^{(l)} = \text{LeakyReLU}(\bar{a}^{(l)T} (z_i^{(l)} || z_j^{(l)})), \quad (2)$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik}^{(l)})}, \quad (3)$$

$$h_i^{(l+1)} = \sigma \left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)} \right), \quad (4)$$

(docs.dgl.ai. (n.d.))

Equation 1 is a linear transformation of the lower layer embedding $h_i^{(l)}$ and $w^{(l)}$ is its learnable weight matrix. Equation 2 computes a pair-wise un-normalised attention score between two neighbours. Here, it first concatenates the z embeddings of the two nodes, where $||$ denotes concatenation, then takes dot product of it and learnable weight vector $\bar{a}^{(l)}$, and applies a LeakyReLU in the end. This form of attention is usually additive attention, contrast with a dot-product attention in the transformer model. Equation 3 applies a SoftMax to normalise the attention scores on each node's incoming edges. Equation 4 is like GCN. The embedding from neighbours is aggregated together, scaled by the attention scores.

As all the GNN models have the same neural network structure as the DNN, they all use similar forward and backwards propagation methods to increase the accuracy of predictions. In the same way as the DNN they use ReLU activation functions, with BCE with logit loss functions and gradient descent and the backpropagation method.

Pre-processing

When applying all six models, preprocessing steps were applied to the input data to ensure it is in the correct format to be modelled. First, the categorical variables had a hot-one encoder applied to them, this turned the categories into numerical values while keeping them grouped in specifically different cases, for example, the variable Sex, had 2 values present, M for male and F for female, instead the hot-one encoder would transform these values into 1 for male and 0 for female, allowing mathematical transformations to be applied but still keeping the information in a categorized format. For numeric variables, a standard scaler was applied, this is a useful tool in machine learning that transforms all the numbered values onto the same scale removing the bias given to the model when dealing with very large and very small numbers, the equation used by the standard scaler is:

Equation 6

$$x' = \frac{x - \mu}{\sigma}$$

X' is the scaled value, x is the original value, μ is the mean of the variable and σ is the standard deviation of the variable. After the variables have been scaled or encoded imputation was applied, meaning that missing values were substituted with synthetic values. In the case of the numeric variables the median of the variable was inputted into the NaNs, in the case of categorical variables the most frequent group was applied. Using the most frequent group allows the imputed values to have the least effect possible on the model output as they will be seen as more generic results. In the case of numeric values there is no studied difference in whether using the mean or the median gives better results, thus I have opted to use the median, as that is the default in the sklearn package.

The GNN models had further preprocessing step as previously described, as the data had to be reconstructed into a non-Euclidean graph structure. To achieve this, we horizontally stacked the numerical and categorical variable together using the NumPy `hstack()` function combining the arrays into columns, creating the nodes. These nodes were then connected through edges created by the common conditions creating neighbourhoods of similar patients. These edges are given weightings through a Tanimoto score. The Tanimoto score is a simple way to measure the similarity between two sets. It quantifies the similarity between two sets by comparing the number of elements they have in common to the total number of unique elements in both sets, ranging from 0 to 1. The equation is given as:

Equation 7

$$T(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Where:

- $|A \cap B|$ is the number of elements common to both sets (the intersection).
- $|A \cup B|$ is the total number of unique elements in either set (the union).

(Vogt, M et al (2020))

Results

Data Correlation

To gain a better understanding of how the variables interact, we can plot a correlation matrix. This shows which variables are more likely to have similarly high or low values in them, proving which variables are strongly or weakly related to each other. When completing this for the patient table we gain this output:

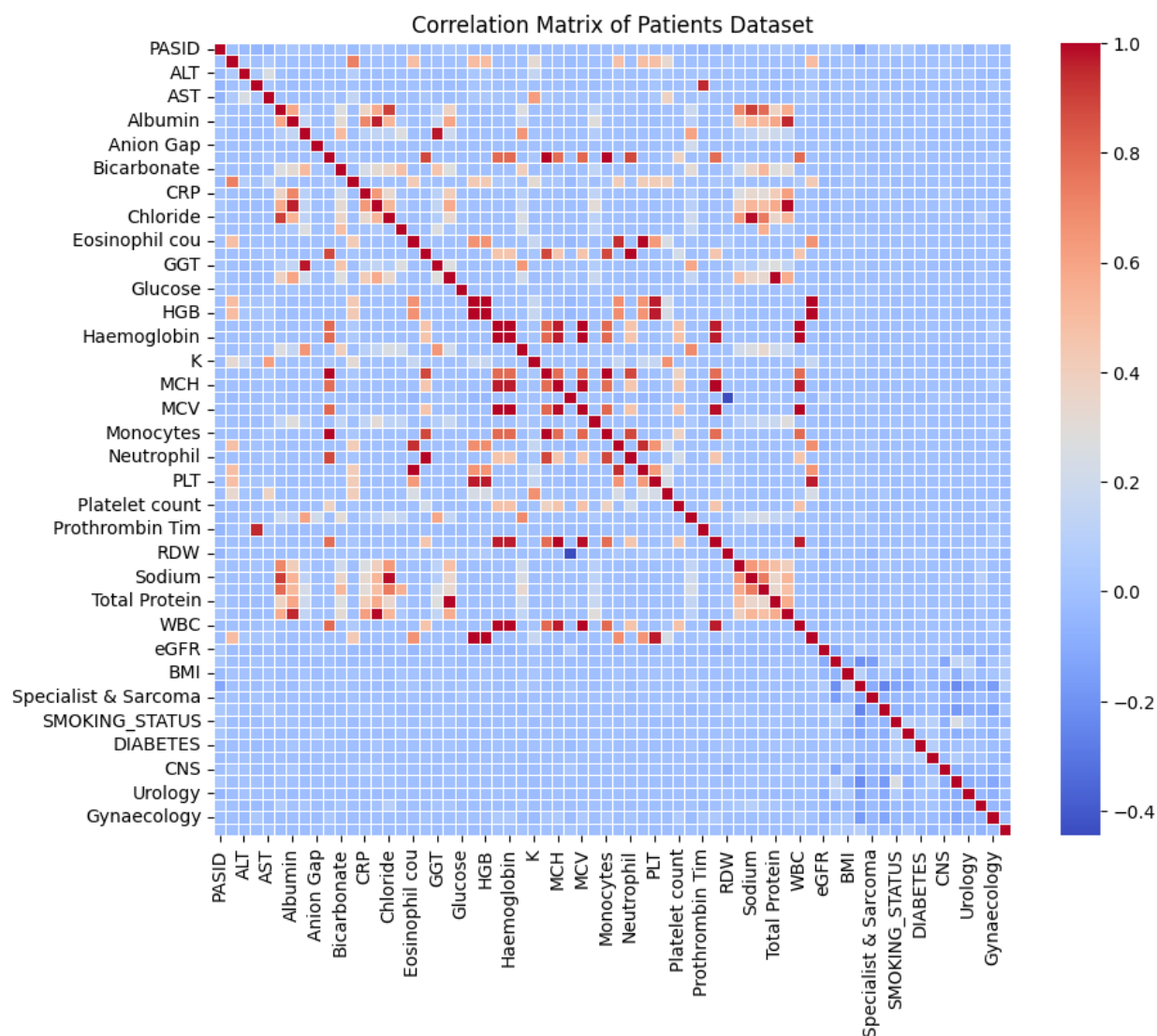


Figure 7

From the key we see that the red colours represent a strong positive correlation, and the blue colours represent strong negative correlation. The plot is split in half, with the diagonal middle rows representing the correlation of a variable with itself, thus in every case having a correlation of exactly 1, this split means that the plot is reflected down this diagonal, as a result only 1 half the plot needs to be examined. Generally, we find that, most variables have no correlation, this is expected as our dataset holds a lot of values that do not necessarily link, suggesting that the dataset holds a broad range of information, which should improve our modelling.

Most correlated variables

- Haematocrit level
- MCV
- WCC
- HGB

- MCH
-

Haematocrit level is the amount of red blood cells in the blood and MCV constitutes the size of the red blood cells and MCH represents the concentration of haemoglobin, while MCH is the mean corpuscular haemoglobin, WCC is the white cell count. These variables make sense to be the most correlated, as they are all general measures of the structure of the blood and thus would correlate with other features about the blood.

Least correlated variables

- Breast
- Lung
- Lower GI
- CNS
- Age

Breast and lung are binary values which represent the presence of breast cancer and lung cancer. These have low correlation with variables as they are showing something very different to the other lab test variables, thus would not have much relation. As in my study, I am looking at many different cancers, you would assume that most patients would have a 0 in both columns which would show low correlation. You would also expect Age to have low correlation as you would expect higher ages to be unhealthier people. Lower GI represents the digestive system, and CNS represents the central nervous system, again these are showing measures of very different things to the blood tests that the other variables represent, thus would have low correlation.

Logistic Regression

The first model implemented to solve this problem was Logistic Regression. In Data Science research Logistic Regression is implemented as a baseline model, as it is the simplest and least expensive model, thus if the more complicated models do not give better results, Logistic Regression should always be used. To make sure the best model for the data from Clatterbridge, a grid search was applied where the best hyperparameters are systematically found by rerunning the model with every combination of the parameters, this is usually very computationally expensive as the model must be rerun continually, but with a very simple Logistic Regression model, this was less of an issue, only taking 11 minutes and 39.8 seconds on the laptop given by Clatterbridge. The hyperparameters used to most distinctly change the Logistic Regression model, the ones used in my grid search are the solver, the optimization algorithm used, C the inverse regularisation strength, the penalty the type of regularisation, and max_iter the number of iterations. The grid search used for this model was:

Solver	liblinear	lbfgs	Newton-cg	Saga	
C	0.01	0.1	1	10	100
Penalty	l1	l2	elasticnet	None	
max_iter	100	200	300		

First, as a test the Logistic Regression model was applied to predict each of the two labels individually, making a model for each one then combining the results. The best parameter combination found in the grid search for the model predicting Toxicity was, solver equals liblinear, meaning it fits a linear boundary for classification, C equals 0.01, giving a large penalisation of the magnitude of the coefficients, resulting in a simpler model, penalty equals l1 regularisation, also known as lasso, where the penalty is the sum of the absolute values of the coefficients times the strength of the regularisation and finally, max_iter equals 100, the lowest, again giving a simpler model and reducing overfitting.

Classifier C: 0.01	max_iter: 100	Penalty: l1	Solver: liblinear
--------------------	---------------	-------------	-------------------

For the model predicting Recurrent, max_iter and C had the same values as the model predicting Toxicity, solver equals saga, as a result the models' coefficients are incrementally changed using a different subset of the dataset each iteration, making it faster and more scalable to larger datasets. Penalty equals l2 meaning the penalty is the sum of the squared coefficients times by the magnitude, discouraging the size of the coefficient distributing the weights more evenly.

Classifier C: 0.01	max_iter: 100	Penalty: l2	Solver: saga
--------------------	---------------	-------------	--------------

When applying these models to the test set and plotting a ROC curve, we achieved these plots:

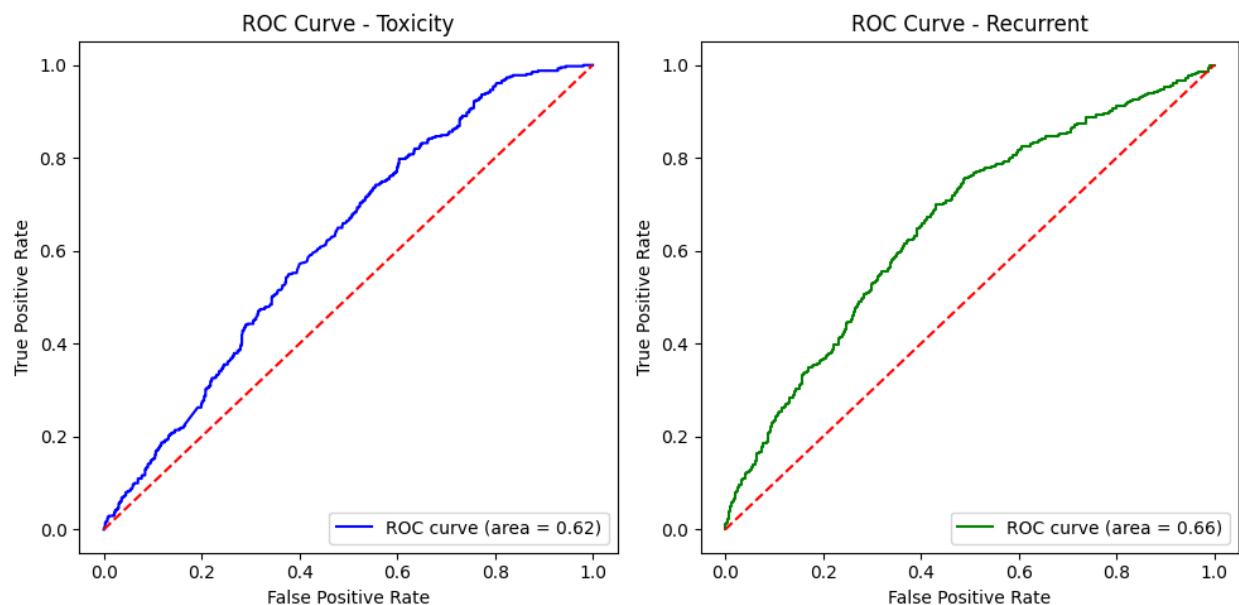


Figure 8

These plots show that the model is predicting Toxicity 62% of the time and Recurrent 66% of the time. As a random guess would theoretically give a 50% AUC score, we find that the logistic regression does mathematically give prediction but has a relatively low AUC score. This

suggests that the data has non-linear relationships, and a more complex model will be necessary to increase predict accuracy.

Constructing the logistic regression into a multiclass format and completing the same grid search gave the solver equal to lbfgs. Meaning instead of calculating the full set of second order derivatives of the boundary, the hessian matrix, the solver approximates the hessian matrix in a method like the Newton Raphson method, making the algorithm more computationally efficient and more robust when transferring to another test set. The max iterations is set to 200, more the model more complex, the magnitude of the weights C is given as 10 and the penalty is given as l2 regularisation.

When applying the Logistic regression in a multiclass fashion we gained this plot:

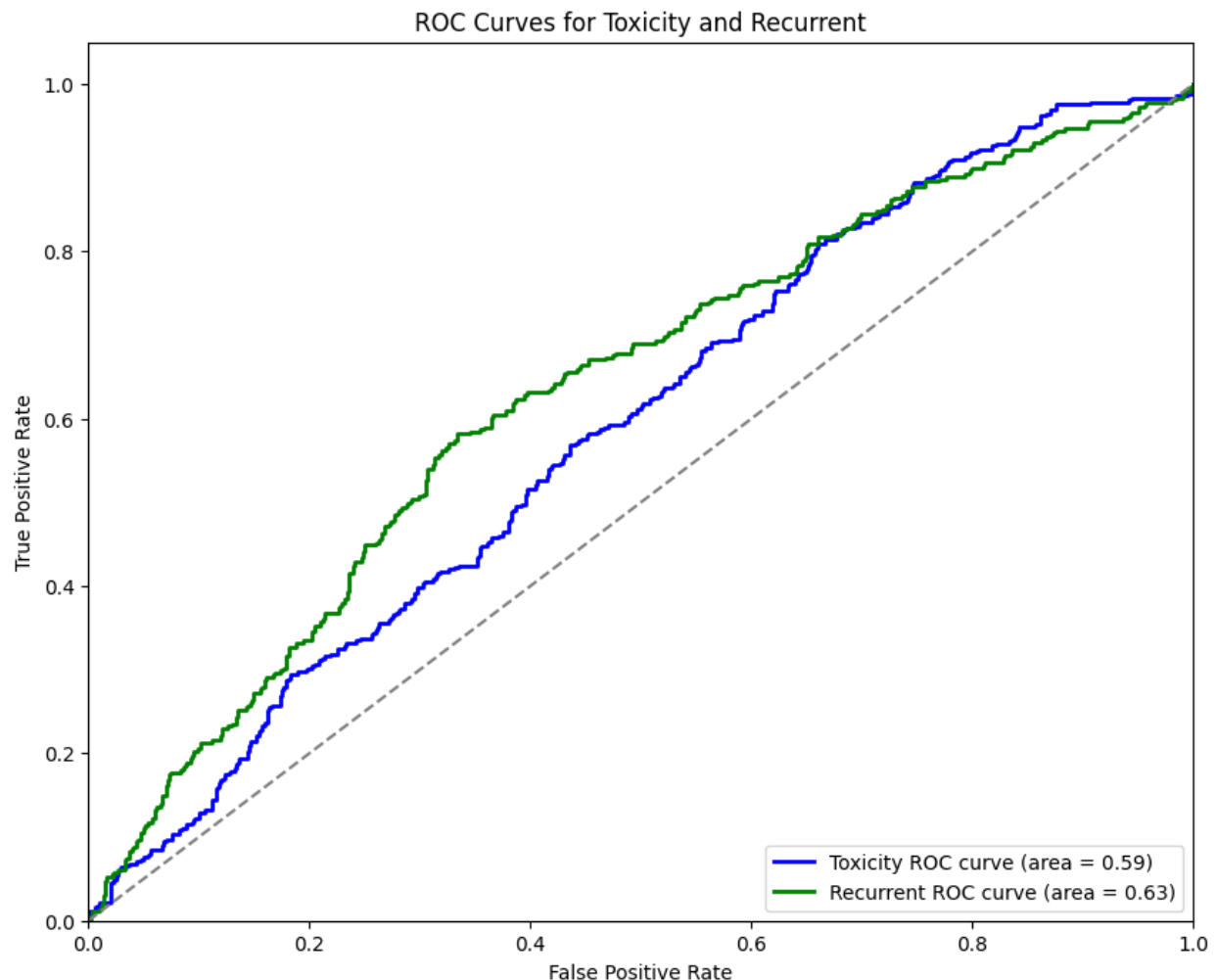


Figure 9

When comparing the multiclass variant to the single label, we find that the multiclass variant has lower accuracies then when individually predicting each class, this is expected as the problem

becomes more complex meaning subtler patterns can be missed and the error is cumulative. However, the difference in AUC score is small and they accuracies are still given as 59% for toxicity and 63% for Recurrence, still beating the 50% we can expect from a random guess.

Instead of predicting the class specifically, we can binarize the classes and predict cases of no adverse effects cases of toxicity, cases of recurrence and cases of both, giving this ROC curve output:

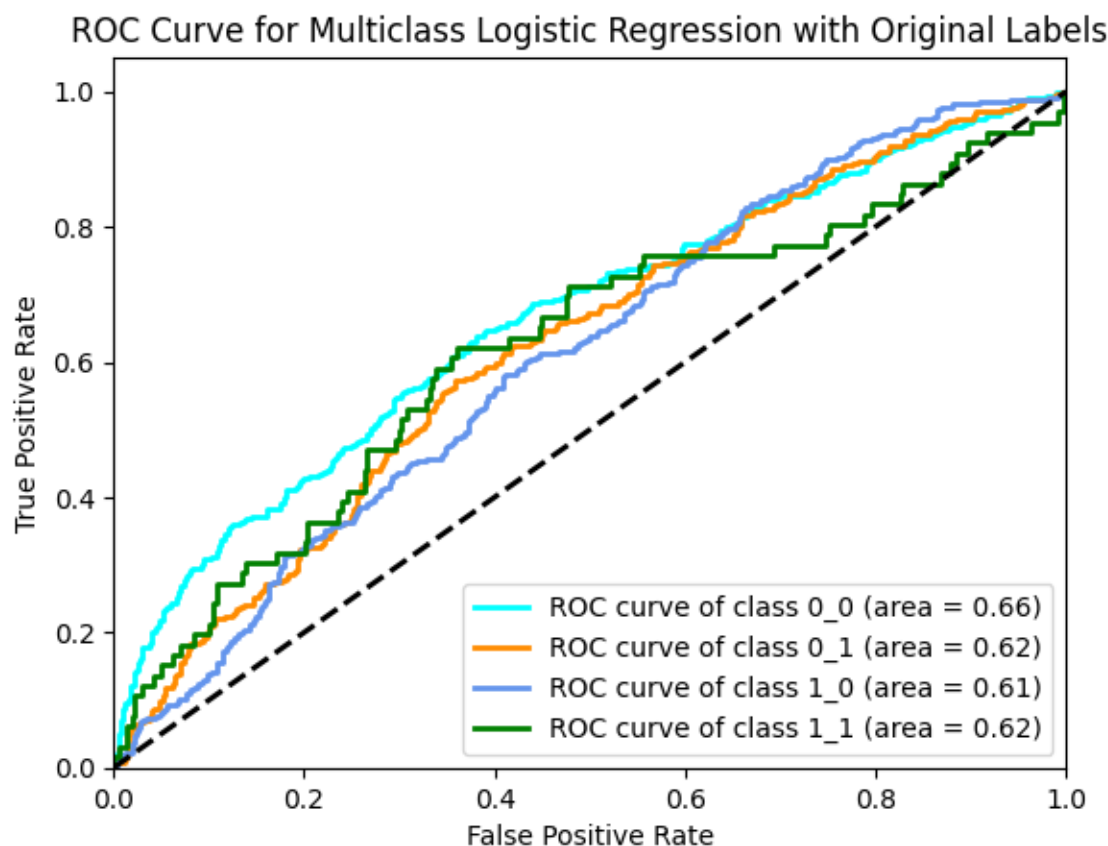


Figure 10

Here we have 0_0 constituting no adverse effects observed, 0_1 only toxicity being observed, 1_0 only recurrent observed and 1_1 both adverse effects observed. We find when performing multiclass classification in binary form, we gain very similar results to the individual prediction with 66%, 62%, 61% and 62% respectively.

In all forms, the accuracy is relatively low suggesting to us that more complex models must be implemented by more complex models.

Random Forests

The next model implemented is Random Forests the hyperparameters being changed in the grid search are, `n_estimators`, denoting the number of trees in the forest, `max_depth`, the maximum depth of said trees, `min_samples_split`, the number of samples required to split an

internal node, min_samples_leaf, the minimum samples required at a leaf node and bootstrap, a binary hyperparameter, denoting whether bootstrap samples are used when building trees.

The grid search used to give the best version of the Random Forest model is:

n_estimators	100	200	300	
max_depth	None	10	20	30
min_samples_split	2	5	10	
min_samples_leaf	1	2	4	
bootstrap	True	False		

Again, to test whether increasing the complex of the model is worth the increase in computational cost, we will perform a Random Forest on the labels individually, to make sure there is an increase in accuracy score.

When looking at the grid search for the model predicting toxicity, we find that n_estimators equal 100, min_samples_split equals 10, min_samples_leaf equals 2, max_depth equals 20 and bootstrapping equals False. Having estimators at 100 makes the model the simplest in the grid search, stopping overfitting by lower the number of individual trees in the model. Having the min_samples_split set to 10 again reduces overfitting but could add bias by preventing the trees growing deep enough to capture intricate patterns in the data. Max_depth having a value of 20 again reduces the complexity of the model, only allowing a maximum of 20 levels in the tree.

When comparing to the model for recurrence, n_estimators, min_sample_leaf and max_depth was all the kept the same, with 100, 2 and 20 respectively. The values that differ were min samples split with 5 instead of 10 and bootstapping which was true instead of false. Lowering the min samples split, reduces overfitting and makes the model more generalisable by ensuring shallower trees. Similarly adding bootstrapping, trains the individual trees on slightly different training set achieving the same outcomes.

Bootstrap: False	Max_depth: 20	Min_sample_leaf: 2	Min_sample_split : 10	N_estimators: 100
---------------------	---------------	-----------------------	--------------------------	----------------------

Bootstrap: False	Max_depth : 20	Min_sample_leaf: 2	Min_sample_split: 5	N_estimators:100
---------------------	-------------------	--------------------	------------------------	------------------

The two ROC curves produced by these models are shown:

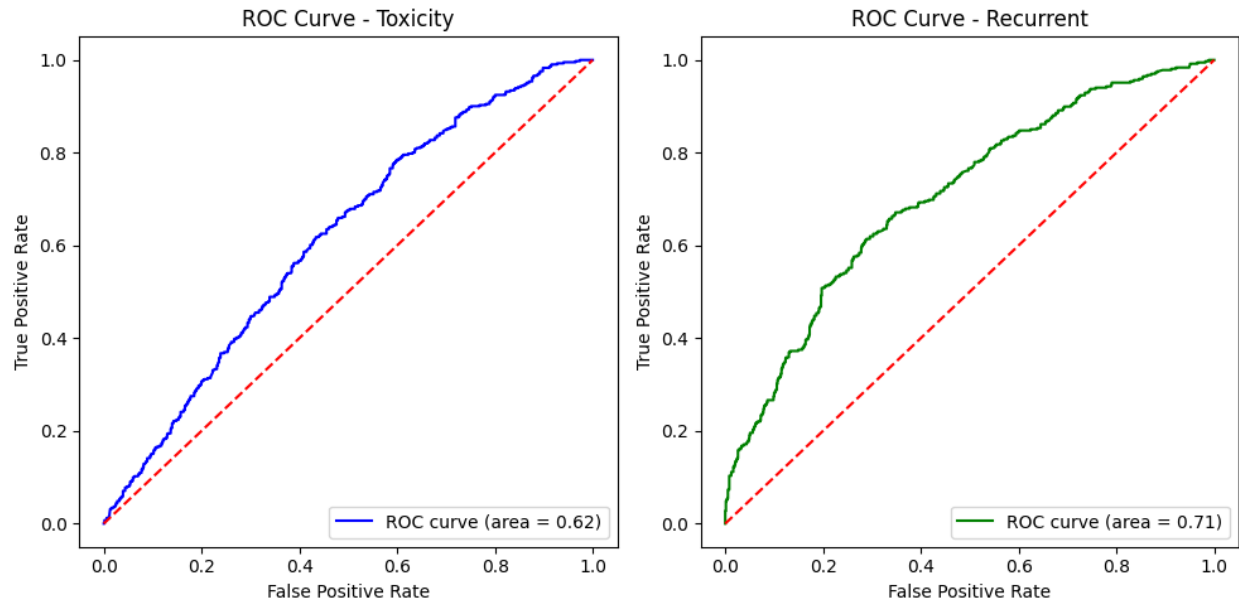


Figure 11

From this we see an improvement on the Logistic Regression model, suggesting that there are more complex patterns in the data that the Logistic Regression cannot find. This means that it is a good idea to proceed in making the more complex modelling. This modelling took 8 minutes and 22.1 seconds to run on the laptop given by Clatterbridge.

The parameters giving the best model for the data given were, `n_estimators` equal to 100, giving the smallest and simplest amount of trees, `max_depth` equals `None`, meaning that each tree will keep splitting until all leaves are pure, they contain only one instance of a single class or leaves cannot split any further, until the min samples split has been achieved, without having an artificial maximum, `min_samples_split` is given as 5 and `min_samples_leaf` is given as 1 and finally bootstrapping is applied, reducing overfitting by training trees on different subsets of the dataset.

Bootstrap: True	Max_depth: None	Min_sample_leaf:1	Min_sample_split: 5	N_estimators: 100
--------------------	--------------------	-------------------	------------------------	----------------------

Using this multi-class model, we achieved this ROC curve plot:

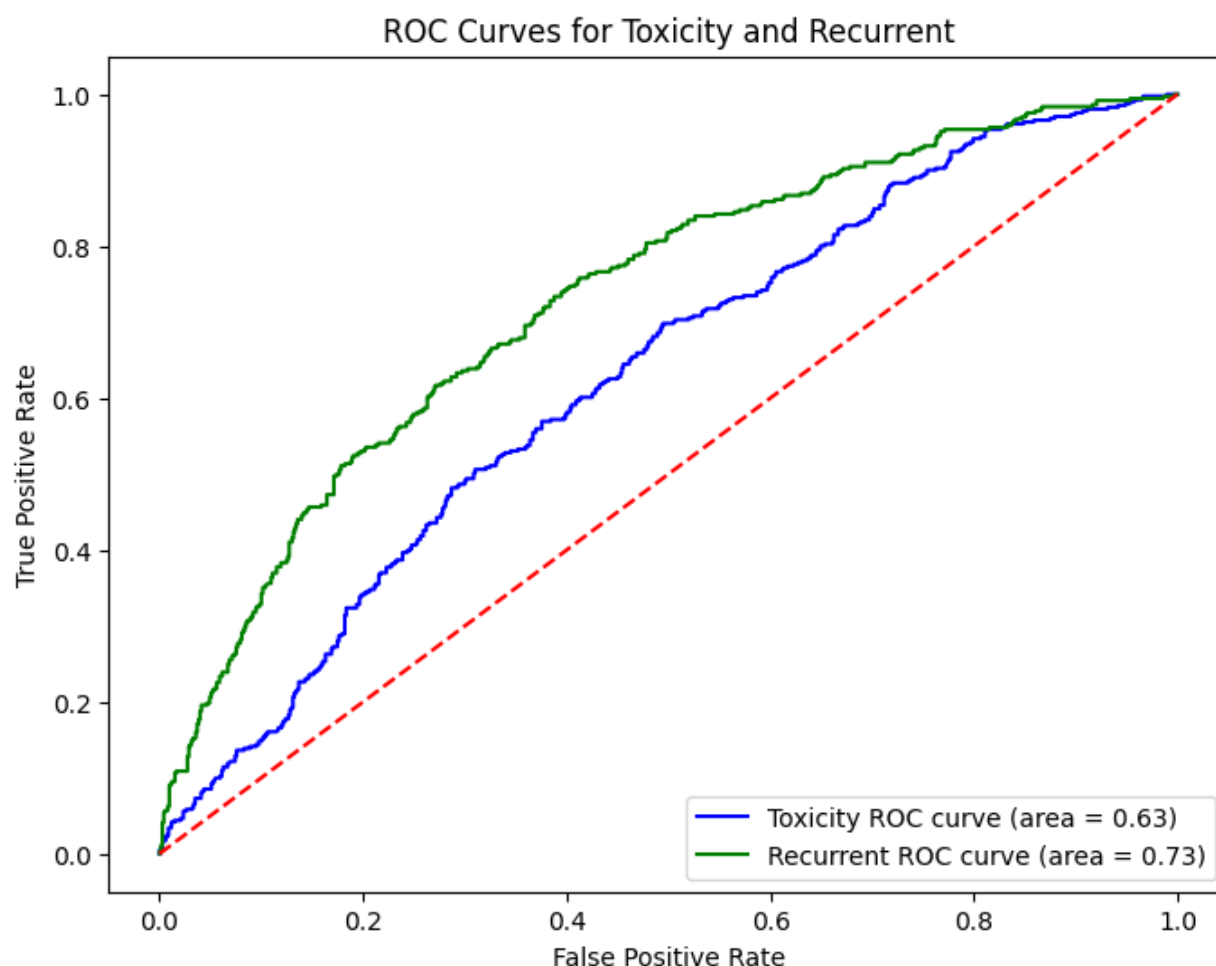


Figure 12

This shows a significant increase over the simple Logistic Regression model, with toxicity having an AUC score of 63% and Recurrence AUC score of 73%, showing an increase in AUC of 4% and 10% respectively. This improvement suggests that the data does indeed have some more complex relationships other than the linear relationships discovered through the Logistic Regression. The model is still not perfect however as both labels did struggle to reach the score we can expect from high performing machine learning models, of $\geq 85\%$ AUC, thus we will need to attempt to create an even more complex model.

Dense Neural Network

When implementing the Deep Neural Network, we used the python package PyTorch instead of sklearn, as sklearn does not have the capabilities to create neural networks. The two packages mainly used for Neural Networks in python are PyTorch and TensorFlow, both frameworks are robust and powerful, so ultimately picking between them becomes personal choice. I chose PyTorch, due to its better error messaging, and because it uses python's class object for the model, it allows extra layers to be added in a more intuitive way than TensorFlow.

As Neural Networks are a collection of neurons, their frameworks are less constricting, having a much more customisable structure. As a result, their features cannot be systematically run through in a grid search. Instead, a trial-and-error process had to be done to find the right type and number of layers to find the best accuracy scores. This does however make creating the perfect neural network very difficult, as one could keep tweaking it gaining different results for a very long time. I started with a very simple DNN with two layers, achieving poor results close to the random guess. Systematically adding more layers gradually improved the model until we gained this this structure:

```
class ImprovedDNN(nn.Module):
    def __init__(self, input_size, output_size):
        super(ImprovedDNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.relu1 = nn.ReLU()
        self.dropout1 = nn.Dropout(0.5)

        self.fc2 = nn.Linear(256, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.relu2 = nn.ReLU()
        self.dropout2 = nn.Dropout(0.5)

        self.fc3 = nn.Linear(128, 64)
        self.bn3 = nn.BatchNorm1d(64)
        self.relu3 = nn.ReLU()
        self.dropout3 = nn.Dropout(0.5)

        self.fc4 = nn.Linear(64, output_size)

    def forward(self, x):
        x = self.dropout1(self.relu1(self.bn1(self.fc1(x))))
        x = self.dropout2(self.relu2(self.bn2(self.fc2(x))))
        x = self.dropout3(self.relu3(self.bn3(self.fc3(x))))
        x = self.fc4(x)
        return x
```

Figure 13

Taking a closer look, we see that the network has 4 layers, 3 of which are complex layers with batch normalisation, ReLU transformations and Dropout, and the final layer being just a Linear output layer. The input layer at the start is made up of 256 neurons and takes in the input data, the batch normalisation layer making the data have 0 mean and 1 unit variance improving the speed and stabilisation of the model. ReLU layer introduces non-linearity making the model perceive more than just linear patterns in the data, and finally a dropout layer is added to prevent overfitting by setting neurons weight to 0. The parameter of 0.5 means that 50% of the neurons will be set to 0, this parameter had to be further tested. On the data we found that 50% and 60% gave joint highest AUC scores. Looking further through the DNN, halving the number

of neurons in each layer, this is standard practice as it helps with regularisation, improves computational efficiency, aligning with intuition as high-level features require fewer dimensions to represent properly. The final layer is just a linear layer as it is not making any prediction calculations, it only is mapping features to the desired output structure. In training this model took around 2 minutes and 32 seconds to train on the Clatterbridge laptop.

The DNN previously described with 0.6 dropout was the same in structure and size, just with the 0.6 as the parameter for the dropout layer instead.

```
class ImprovedDNN(nn.Module):
    def __init__(self, input_size, output_size):
        super(ImprovedDNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.relu1 = nn.ReLU()
        self.dropout1 = nn.Dropout(0.6)

        self.fc2 = nn.Linear(256, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.relu2 = nn.ReLU()
        self.dropout2 = nn.Dropout(0.6)

        self.fc3 = nn.Linear(128, 64)
        self.bn3 = nn.BatchNorm1d(64)
        self.relu3 = nn.ReLU()
        self.dropout3 = nn.Dropout(0.6)

        self.fc4 = nn.Linear(64, output_size)

    def forward(self, x):
        x = self.dropout1(self.relu1(self.bn1(self.fc1(x))))
        x = self.dropout2(self.relu2(self.bn2(self.fc2(x))))
        x = self.dropout3(self.relu3(self.bn3(self.fc3(x))))
        x = self.fc4(x)
        return x
```

Figure 14

As the problem the network is solving is a multi-class problem, we used a BCE with logit loss as the loss function. This loss function takes the raw inputs logits, applies a sigmoid function internally and computes binary cross-entropy loss. It had one parameter of `pos_weight = class_weights`, this means that pos weight allows the model to weigh the positive samples larger than the negative ones, which is helpful because we have more negative instances than positive. Class weights also give more weight to the lower classes, I have 2120 cases of toxicity and 1805 cases of recurrence, with 1565 no adverse effects, as the groups are slightly imbalanced, the class weights will assure the modelling can find intricate patterns in the lower classes.

The optimiser used is a RMSprop, meaning Root Mean Square Propagation, it is an adaptive learning rate optimiser that adjusts the learning rate based on a moving average of the squared gradients. RMSprop helps prevent large oscillations in the weights and improves convergence, particularly in deep networks with high non-convex loss surfaces. The optimiser was given a learning rate of 0.001, which is relatively low, meaning that the optimiser will adjust the models' parameters in steps of 0.001, which helps the model converge smoothly to a good solution without overshooting the optimal point.

```
# Define loss function for multi-label classification with class weights
criterion = nn.BCEWithLogitsLoss(pos_weight=class_weights)

# Define optimizer
optimizer = optim.RMSprop(model.parameters(), lr=0.001)
```

Figure 15

The output ROC curve given for the first model with dropout 0.5:

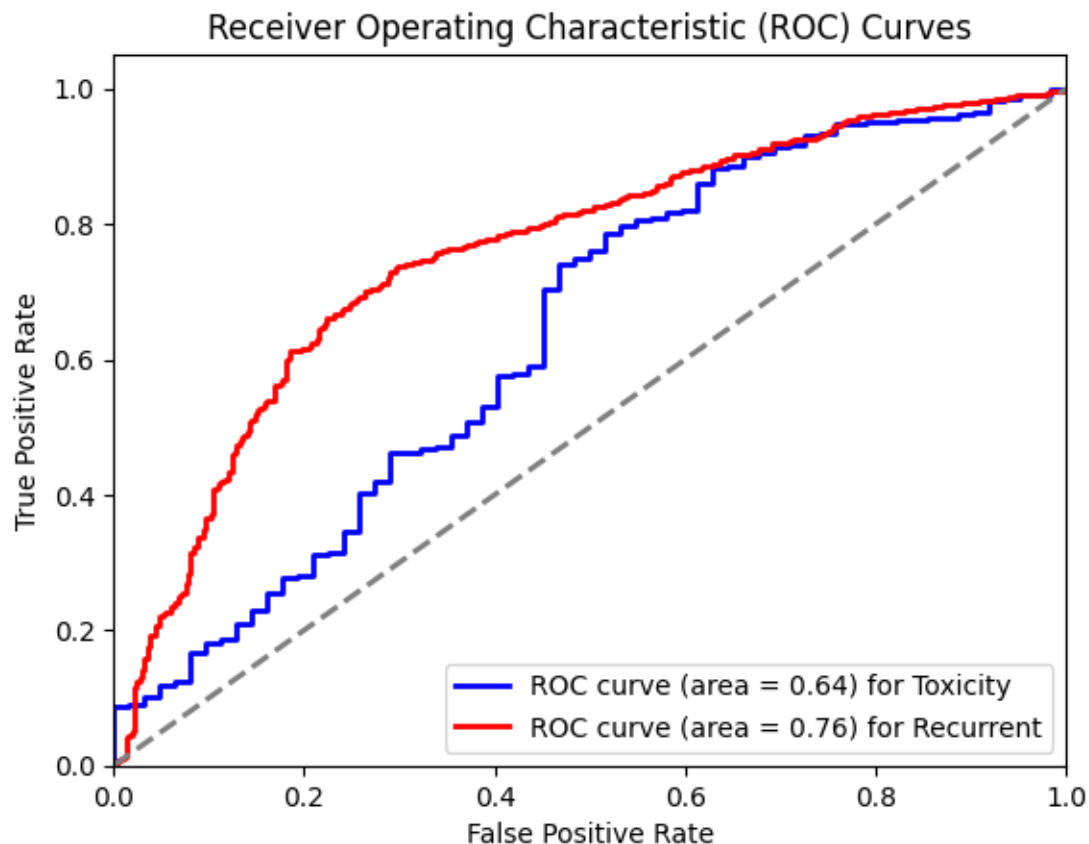


Figure 16

Again, we see a greater improvement over the simpler modelling, suggesting complexity in our data. Overall, this model had an AUC score of 70% a vast improvement over the 25% given by a random guess. Toxicity specifically had an AUC of 64% and recurrence had an AUC of 76%. This means that the model is good at predicting recurrence, but again struggles more on

toxicity. From other studies we should see an improvement on the AUC score when using the GNN modelling.

Comparing to the ROC output of the model with dropout 0.6:

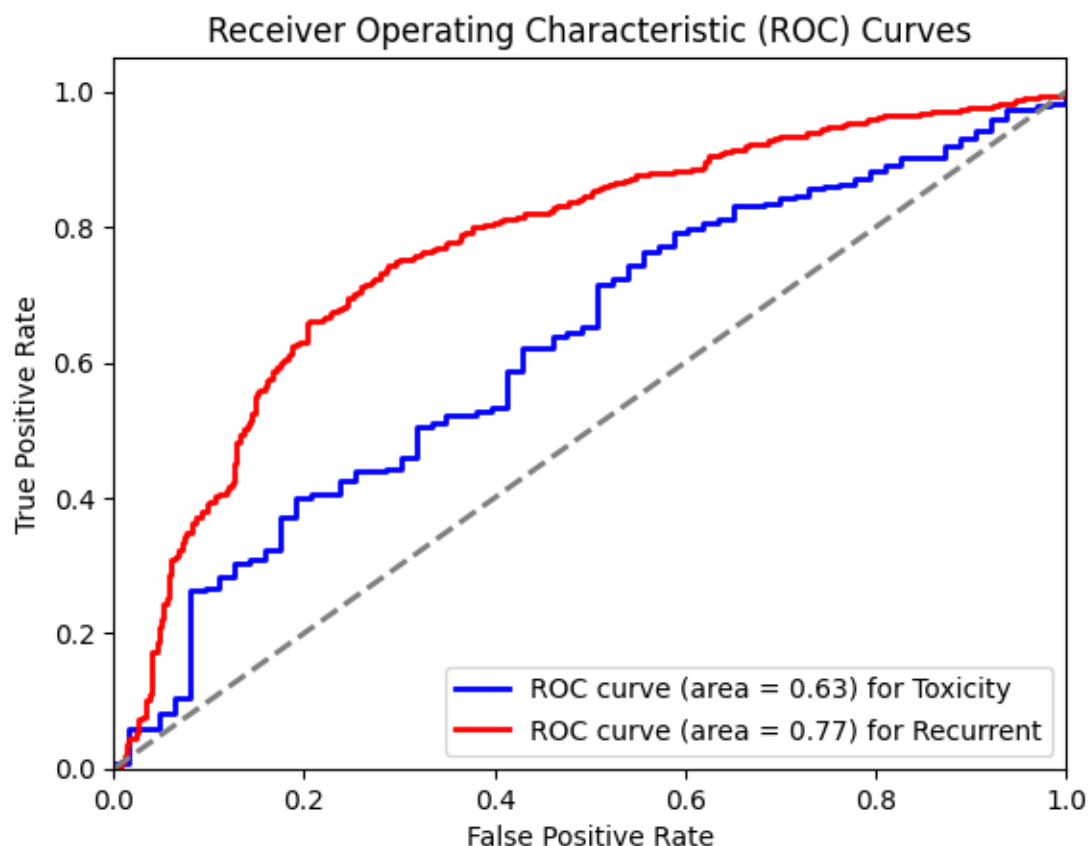


Figure 17

As described before, the model with dropout 0.6 had the same overall AUC score of 70%, however it has slightly worse at predicting toxicity and slightly better at predicting recurrence, thus if you wanted to be more certain about the recurrent prediction you could use this model instead, or vice versa; the difference in AUC score is very minimal at only 1%, and could be written down to variance and error.

Simple Graph Convolution

The first and simplest model implemented on the constructed graph data was the SGC model. As previously described the simple graph convolution network does not have multiple convolution layers as the whole point is to collapse the multiple layers into a singular linear transformation. As a result, the structure is less diverse, only having 1 convolution layer, in most cases. The structure of my model is as follows:

```
# Define model with increased complexity and non-linearity
class EnhancedSGC(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim=2, K=2):
        super(EnhancedSGC, self).__init__()

        # SGC layer
        self.conv = SGConv(input_dim, hidden_dim, K=K, cached=True)
        self.hidden = nn.Linear(hidden_dim, hidden_dim) # Additional linear layer
        self.output_layer = nn.Linear(hidden_dim, output_dim)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        # SGC layer
        x = self.conv(x, edge_index)

        # Non-linearity and additional linear layer
        x = torch.relu(x)
        x = self.hidden(x)
        x = torch.relu(x)

        # Output layer
        x = self.output_layer(x)

        return x # Logits for each class

# Instantiate the model with your desired dimensions
input_dim = x_combined.shape[1] # Number of input features per node
hidden_dim = 128 # SGC typically uses a single hidden dimension
output_dim = 2 # Number of output labels
```

Figure 18

Again to help the model gain better prediction accuracy, to accompany the convolution layer, a ReLU layer was added to either side, allowing the model to capture non-linear, more complex patterns in the data. We used 128 hidden dimensions meaning each node in the graph is

represented by a vector of length 128, again, allowing for more complexity in the representation. As this is a very simple model it took 5 seconds to train.

We also used the same criterion as in the DNN with `pos_weight = class_weights`. The optimiser used is the ADAM optimiser, it combines the advantages of two other popular optimization algorithms: AdaGrad (which works well with sparse gradients) and RMSProp (which works well in non-stationary settings). Adam provides efficient gradient-based optimization, especially for problems involving large datasets or parameters.

A scheduler was also used in the SGC, we used StepLR adjusting the rate of the optimiser by a factor of gamma for the number of times in steps, we used gamma of 0.1 and steps of 50. This type of learning rate scheduler is used to gradually reduce the learning rate during training, which can help improve convergence and prevent overshooting minima in the loss landscape.

```
# Define your criterion with class weights
criterion = nn.BCEWithLogitsLoss(pos_weight=class_weights)

# Define optimizer with L2 regularization (weight decay) and learning rate scheduler
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=5e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.1) # Adjust step_size and gamma as needed
```

Figure 19

When plotting the ROC curve for this model we achieved this output:

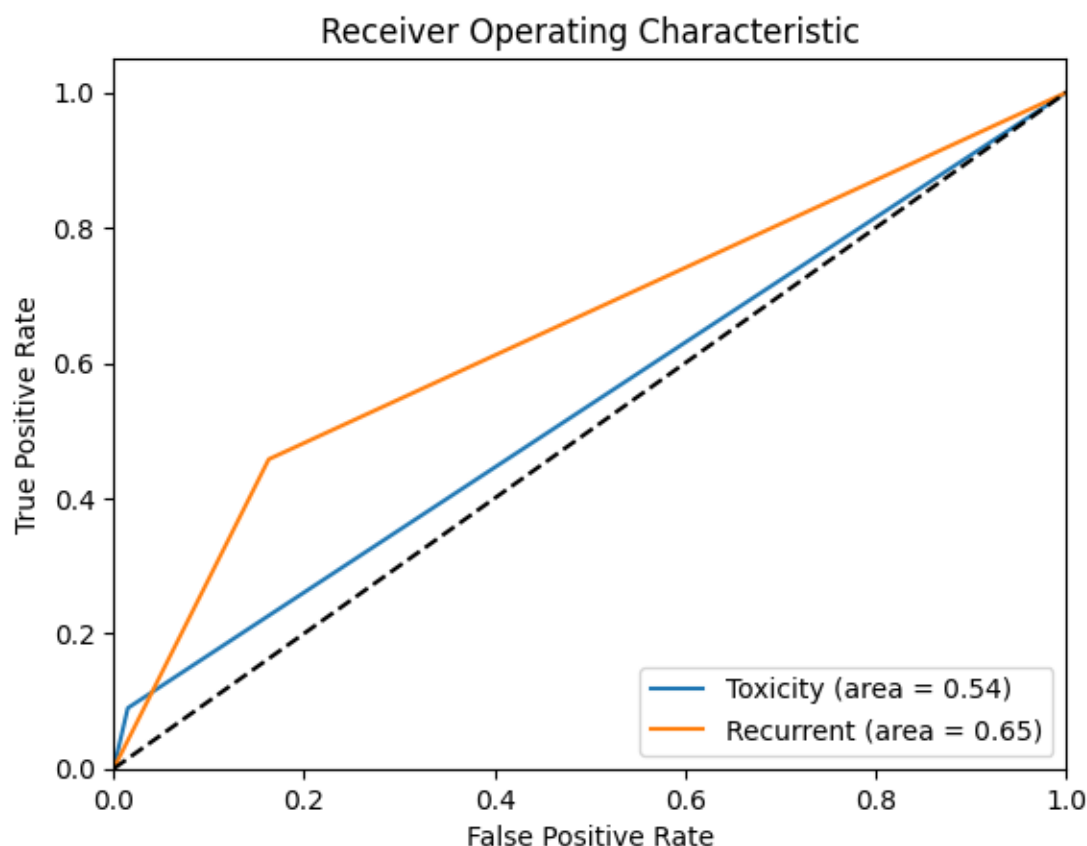


Figure 20

We gain an overall AUC of 60%, with a toxicity AUC of 54% and a recurrent AUC of 64%. Again, in a similar pattern to the previous models, we see that the model struggles predicting Toxicity and does better on recurrent. Toxicity prediction is noticeably worse than the more complicated standard machine learning models, having just about the random guess. The poorer results are expected as this model is the simplest of the GNN models.

Graph Convolution Network

The next GNN model implemented was the GCN, the standard Graph Convolution Network. Just like the DNN model, this model's structure could be change and added to in any way you wanted thus, multiple structures were tried and tested. After this process the model with the best accuracy had this structure:

```

# Define the GCN model
class GCN(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.conv3_toxicity = GCNConv(hidden_dim, 1) # Output for Toxicity task
        self.conv3_recurrent = GCNConv(hidden_dim, 1) # Output for Recurrent task

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = self.conv2(x, edge_index)
        x = torch.relu(x)
        out_toxicity = self.conv3_toxicity(x, edge_index) # Output for Toxicity
        out_recurrent = self.conv3_recurrent(x, edge_index) # Output for Recurrent
        return out_toxicity.squeeze(), out_recurrent.squeeze()

# Assuming graph_data contains x, edge_index, y (toxicity and recurrent labels), and masks
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

input_dim = graph_data.x.shape[1]
hidden_dim = 128

# Instantiate model and move to device
model = GCN(input_dim, hidden_dim).to(device)

```

Figure 21

As you can see, this model has 3 convolution layers, meaning the node neighbourhoods are aggregated 3 times. The aggregation function uses a summing junction to learn the characteristics of the neighboured, for example, if a neighbourhood of three nodes had values 1, 2, 3, that aggregated neighbourhood would have the values 6. In between the convolution layers a ReLU function was applied to add non-linearity to the model. Finally, the output layer had to be split between the two predicted labels. This was not done for any mathematical reason; it was a way of fixing issues in the code of the model. The model also used 128 hidden dimensions; this was tried at higher values, but the model started too overfit. As well as the hidden dimensions, more convolution layers were also tried however, again, the model started to overfit the data, leading to worse AUC scores.

As this model is much more mathematically complex then the other models, it was much more computationally expensive, to help with this issue the device line of code was added this forces the code to be run on the computers GPU, which should help this computation. This model took 18 minutes and 1 second to run.

Again, a BCEwithlogitloss criterion was used with a ADAM optimiser:

```
# Loss functions
criterion_toxicity = nn.BCEWithLogitsLoss()
criterion_recurrent = nn.BCEWithLogitsLoss()

# Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)
```

Figure 22

Using this model, we gained this output:

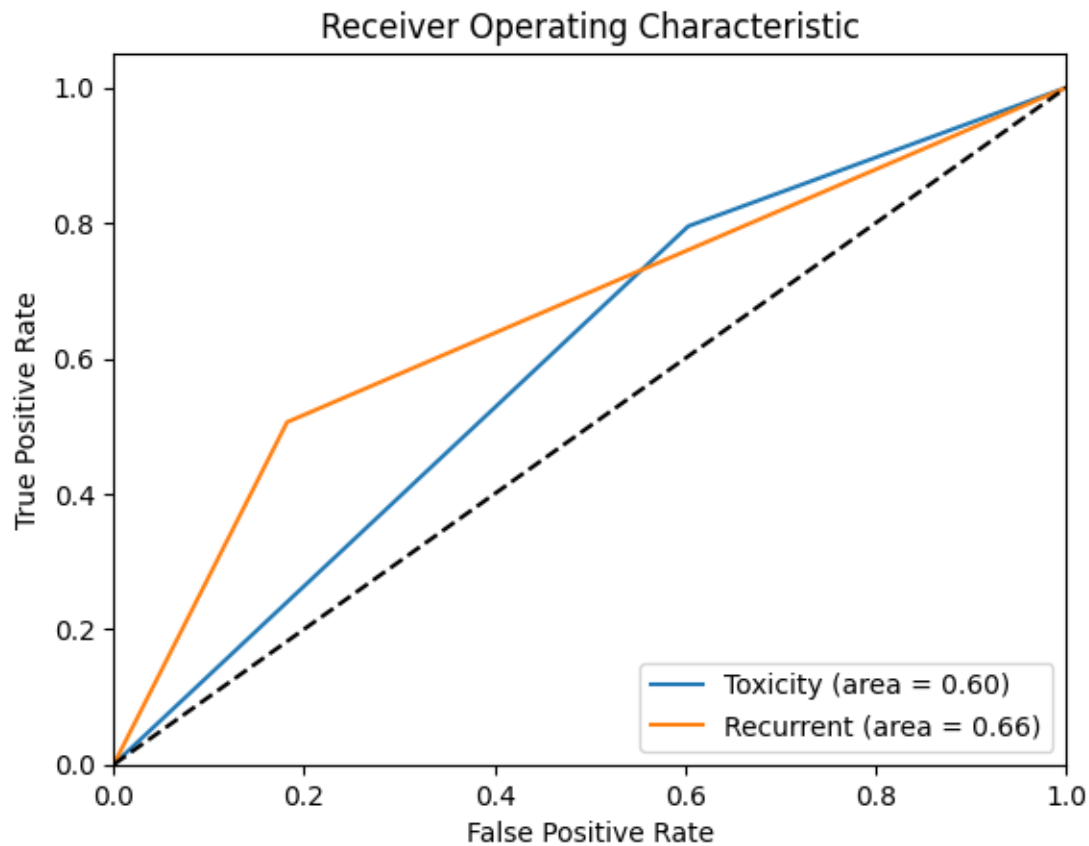


Figure 23

Gaining an overall AUC of 63% with 60% on toxicity and 66% on recurrence. Again, this AUC score could be better suggesting that the model should be made more complex to get better scores, however when more convolution layers were added to our data, the model started too overfit. This means that we need a more complex dataset to gain more accurate model, this could be in more samples of both toxicity and recurrence or gaining more descriptive lab tests and patient demographics.

Using 256 hidden dimensions gave us this model:

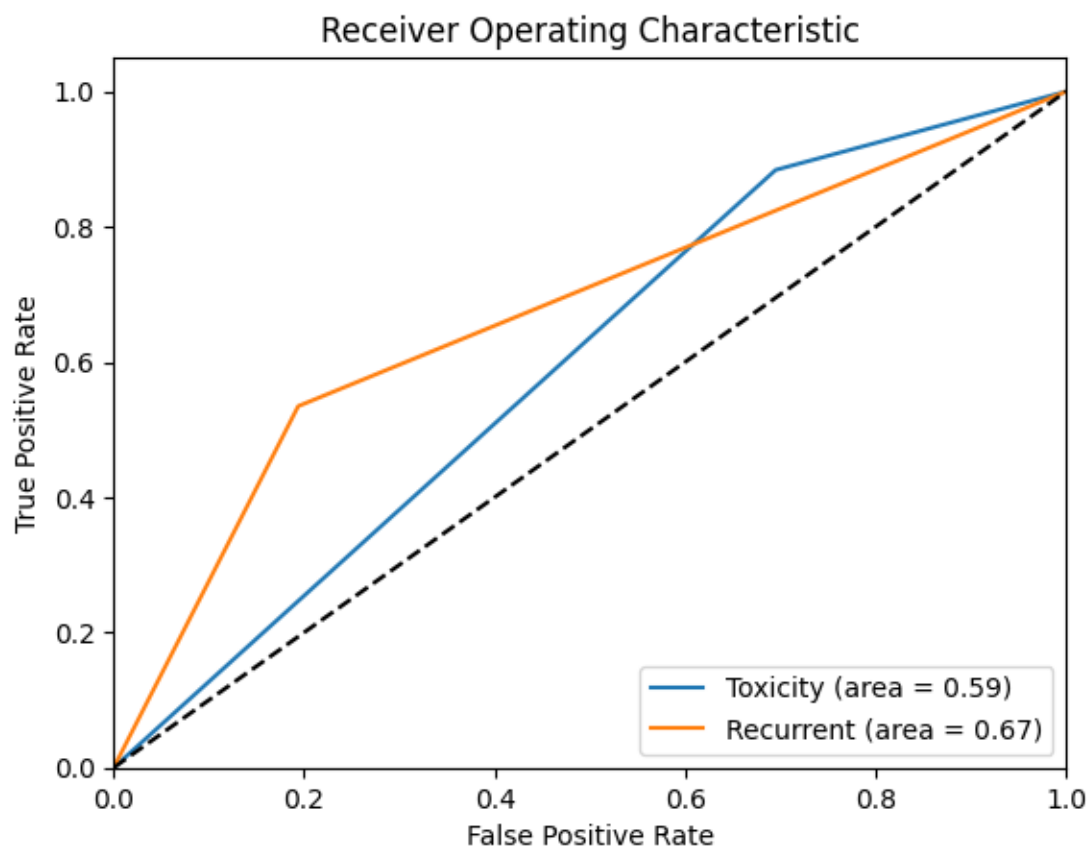


Figure 24

This model had the same overall AUC 63%, but did better at predicting recurrent over toxicity, meaning both models could be used in tandem depending on which label you want to predict with more accuracy.

Graph Attention Network

Finally, a GAT model was implemented on the data. This model is even more computationally taxing than the standard GCN model, due to the introduction of context transformers, as a result the structure of the model had to be carefully fine-tuned like the GCN to make sure the model did not overfit the model, but still produce good prediction accuracy. In practice, when trying to run the model on the laptop given to us by Clatterbridge, even though it is a powerful computer with a lot of memory we gained this error:

```
RuntimeError: [enforce fail at alloc_cpu.cpp:114] data. DefaultCPUAllocator: not enough memory: you tried to allocate 10014388224 bytes.
```

Figure 25

To fix the issue I got in contact with the IT department to give me remote access to the server allowing me to access to 64GB of memory allowing me to be able to run the model.

The model I found to give the best AUC score was:

```

class EnhancedGAT(nn.Module):
    def __init__(self, input_dim, hidden_dim1, hidden_dim2, hidden_dim3, output_dim=2, num_heads=4):
        super(EnhancedGAT, self).__init__()
        self.conv1 = GATConv(input_dim, hidden_dim1, heads=num_heads, concat=True)
        self.norm1 = BatchNorm(hidden_dim1 * num_heads)
        self.dropout1 = nn.Dropout(p=0.4)
        self.conv2 = GATConv(hidden_dim1 * num_heads, hidden_dim2, heads=num_heads, concat=True)
        self.norm2 = BatchNorm(hidden_dim2 * num_heads)
        self.dropout2 = nn.Dropout(p=0.4)
        self.conv3 = GATConv(hidden_dim2 * num_heads, hidden_dim3, heads=num_heads, concat=True)
        self.norm3 = BatchNorm(hidden_dim3 * num_heads)
        self.dropout3 = nn.Dropout(p=0.4)
        self.conv4 = GATConv(hidden_dim3 * num_heads, hidden_dim3, heads=1, concat=False)
        self.norm4 = BatchNorm(hidden_dim3)
        self.dropout4 = nn.Dropout(p=0.4)
        self.output_layer = nn.Linear(hidden_dim3, output_dim)

```

Figure 26

```

def forward(self, data):
    x, edge_index = data.x, data.edge_index
    x = self.conv1(x, edge_index)
    x = F.elu(x)
    x = self.norm1(x)
    x = self.dropout1(x)
    x = self.conv2(x, edge_index)
    x = F.elu(x)
    x = self.norm2(x)
    x = self.dropout2(x)
    x = self.conv3(x, edge_index)
    x = F.elu(x)
    x = self.norm3(x)
    x = self.dropout3(x)
    x = self.conv4(x, edge_index)
    x = F.elu(x)
    x = self.norm4(x)
    x = self.dropout4(x)
    x = self.output_layer(x)
    return x

```

Figure 27

This model had 4 layers of convolution, increasing the complexity when compared to the GCN. In between the convolution layers, I added ReLU layers, much like in all the other Neural Network adding non-linearity. In addition, Batch Normalisation layers were added after the ReLU layer, helping conversion by renormalising each layer, this is important as it combats a common effect of internal covariant, meaning the activation of the layer can sometimes change the distribution of the of the output values, slowing learning process and conversion. Finally, a Dropout layer of 0.4 was added, this value was tested and found to give the highest accuracy. The model uses 64 hidden dimensions, this is relatively low when compared to the other neural networks to try and reduce the overall complexity of the model.

Despite theoretically being the model that should give the best results of all, the model gave this output:

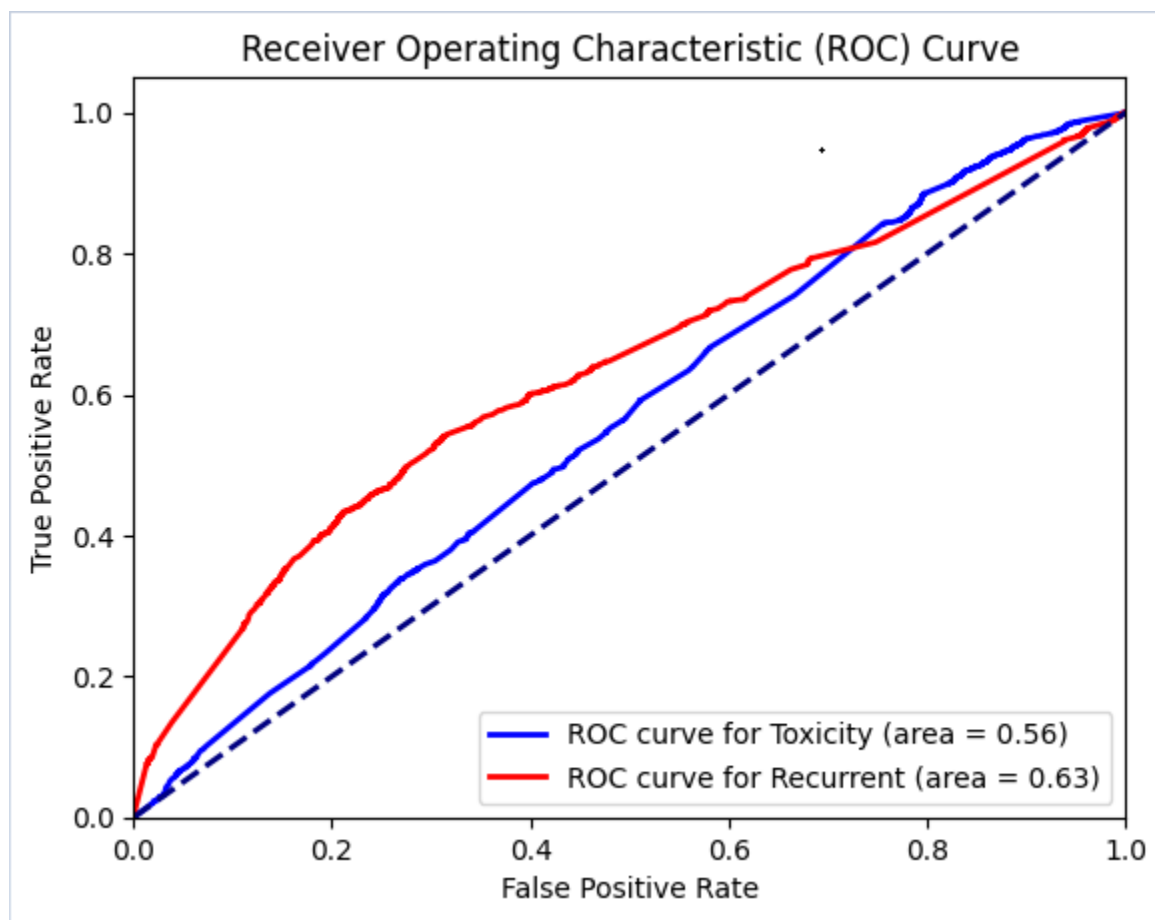


Figure 28

This model gained an overall AUC of 60%, with a toxicity AUC of 56% and a recurrent AUC of 63%. These are disappointing results, much less than what we can expect from the literature review. This is suggesting that the data I extracted from the Clatterbridge server is not complex enough to full utilise the attention transformer model. I suggest, to gain a greater AUC score from the data I should add more samples of each of the labels and add more variables of both comorbidities (edges) and patient's data nodes.

Name	Toxicity	Recurrent	Overall	Time Taken
Logistic Regression	59%	63%	61%	11m 39.8s (GS)
Random Forest	63%	73%	68%	8 m 22.1s (GS)
DNN	64%	76%	70%	2 m 32s
SGC	54%	65%	60%	< 1m
GCN	60%	66%	63%	18m 1s
GAT	56%	63%	60%	100m 38s

Feature Importance

As the DNN was the most accurate model, we can take a closer look at the most important features and see how they individually affect the prediction outcome. We will do this through the SHAP waterfall plot and the SHAP bee swarm plot. For both these graphs types the plots were created from each label individually and the top 20 features from both were plotted.

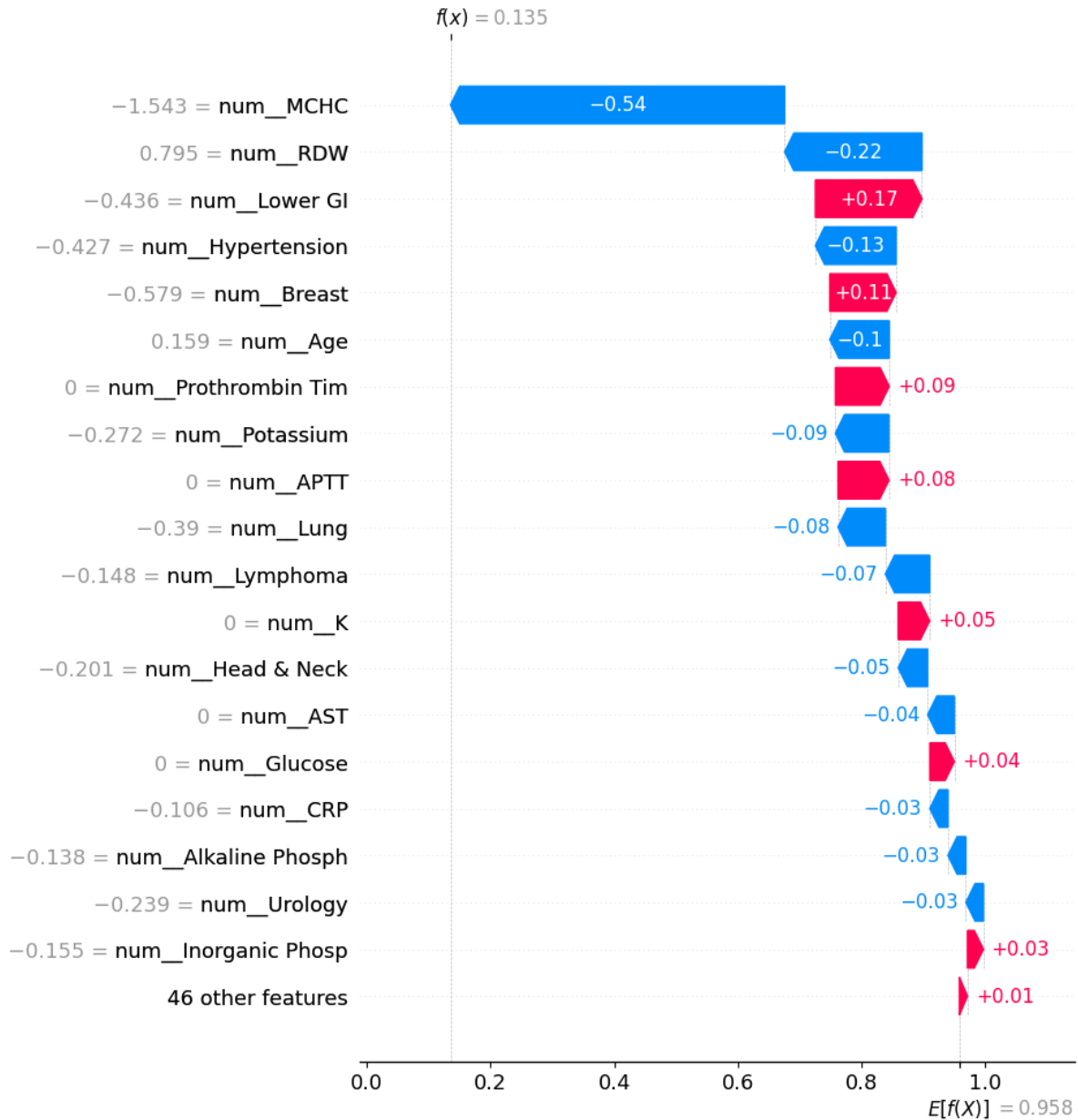


Figure 29

First is the waterfall plot for toxicity. At the bottom of the graph $E[f(X)]$ denotes the average prediction of the model, in our case it is given as 0.958, this means that if the model was given no feature information the model is predicting a near positive case. As a result, we can expect the features to add negatively to the outcome value. The waterfall plot shows how much either positively or negatively each feature changes the outcome value per unit, blue values contribute negatively, and red values contribute positively. We see that for toxicity the 5 features that affect the outcome the most are MCHC with -0.54 , RDW with -0.22 , Lower GI with $+0.17$, Hypertension with -0.13 and Breast with $+0.11$. As expected, the largest contributors give negativity to the outcome value giving a final prediction value of the model being $f(x) = 0.135$.

The bee swarm plot for toxicity is given as:

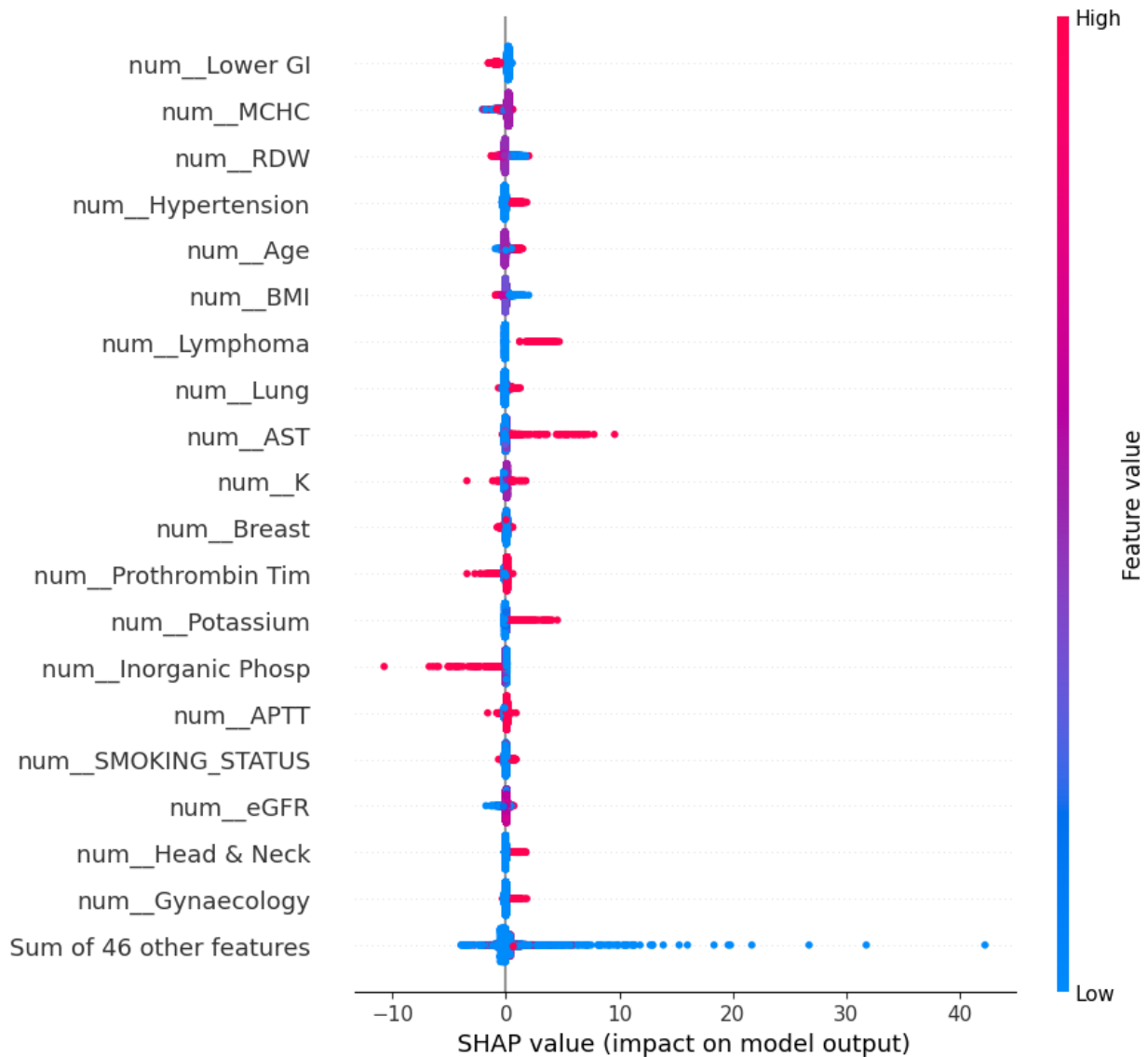


Figure 30

The bee swarm plot is more complicated than the waterfall plot, showing how the individual values effect the outcome prediction. Again, the feature importance in given on the left-hand side, the SHAP values denote the amount of effect the values have had on the prediction outcome, a positive SHAP value shows that the value had a positive impact on the outcome value and a negative show a negative impact. The colour of the dot as shown on the right shows the size of the individual value, blue denotes a small value in the column, red a high value and purple a value around the mean. As expected, the purple values all have SHAP values around 0 having little effect on the outcome. We see those high values in Lower GI, RDW, K, Inorganic_phosp, APTT and smoking_status all effect the model negatively, whereas high values in Hypertension, Age, Lymphoma, AST, K, Potassium, Head & Neck, Gynaecology all effect the model positively. In some features high values effect the prediction negatively and positively, highlighting the non-linearity of the data, explaining the worse results in Logistic Regression. We also see that the lower values generally gave SHAP values close to 0, apart

from in certain features such as RDW and BMI where they both increased the prediction positively and in eGFR where it effected the prediction negatively.

The waterfall plot for recurrence was given as follows:

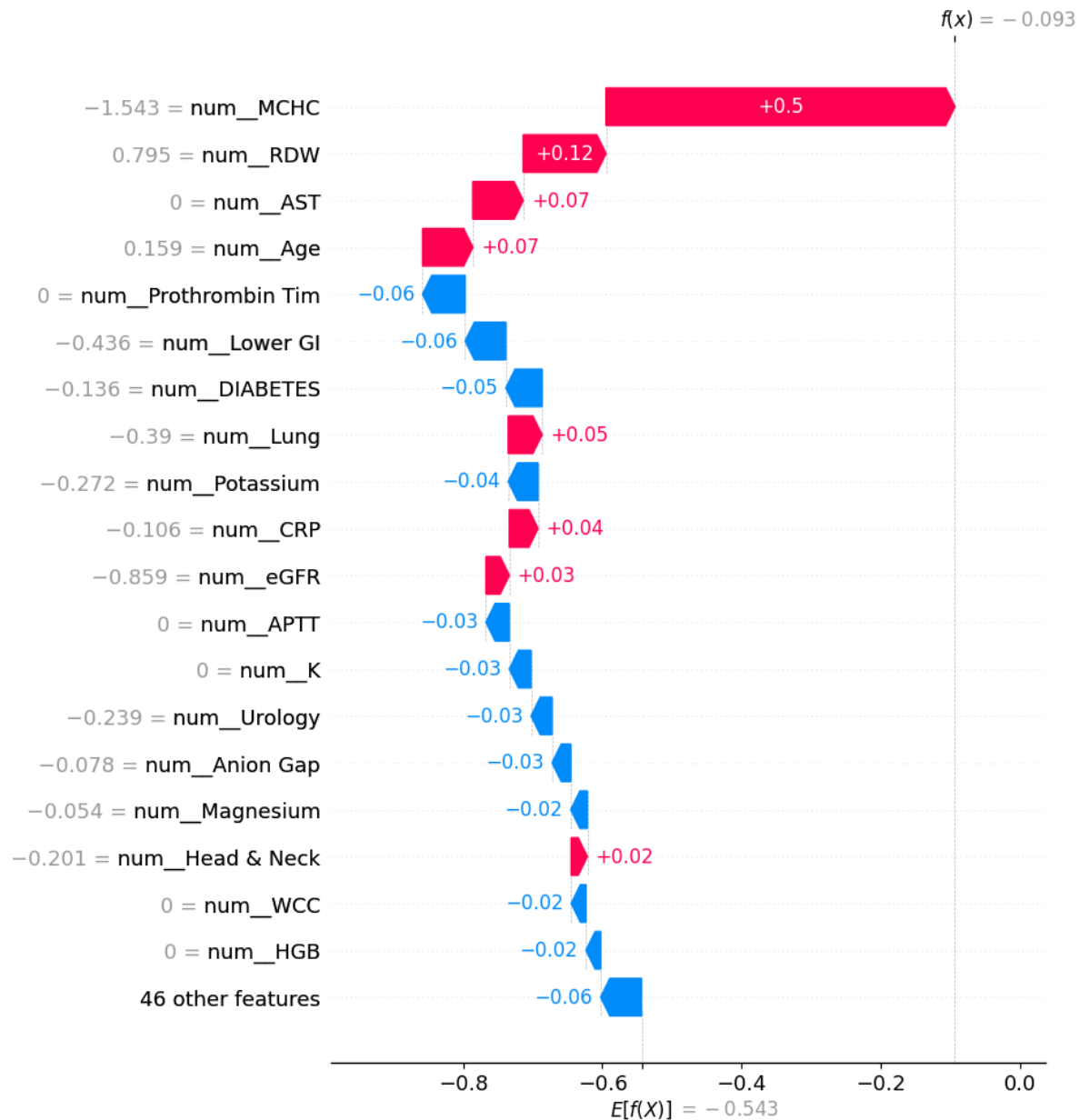


Figure 31

In this case we see that the $E[f(x)] = -0.543$ without any feature input and with the cumulation of all the features the average value became -0.093 . The negative outcome value shows it is predicting a 0 for recurrent, this is expected as the in the outcome data there are less recurrent samples then toxicity. In this case we see that the top 5 most important features are given as MCHC with $+0.5$, RDW with $+0.12$, AST with $+0.07$, Age with $+0.07$ and Prothrombin Tim with -0.06 .

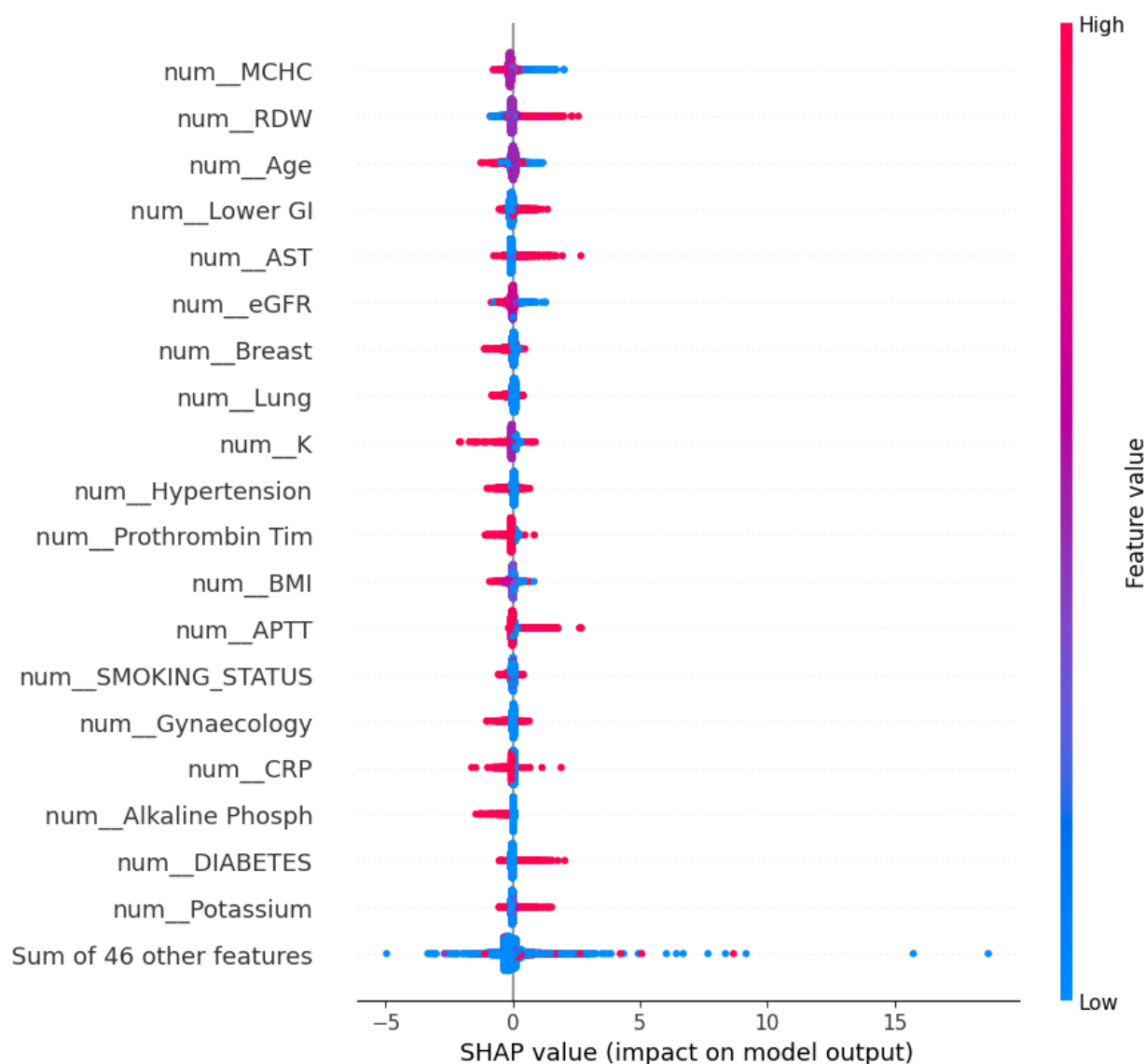


Figure 32

The bee swarm plot for recurrent, shows that the top 20 features show more extreme SHAP values than the toxicity, showing that the individual features have more effect on the prediction outcome, this could be due to the highest accuracy scores of the label. We see that the high values of RDW, Lower GI, AST, K, Hypertension, APTT, Gynaecology, CRP, diabetes and potassium all interact positively with the outcome, whereas the high values of Age, AST, Breast, Lung, K, hypertension, Prothrombin, Gynaecology and CRP all interact negatively with the outcome value. The low values interact positively in MCHC, Age, eGFR, in contrast the only feature where low values negatively impact is in RDW, in all other cases it has SHAP values close to 0.

Conclusions and Further Work

When comparing all six of the models, we find that the DNN was the best model, having the best overall AUC of 70% as well as the best individual scores for both toxicity and recurrence with 64% and 76% respectively, thus this is the model I would recommend using. The second-best model was the Random Forest model with 68% overall and 63% and 73% on the individual labels, highlighting the issue I have faced in this report. As the more complicated models implemented started to overfit the data, I have extracted from Clatterbridge, it suggests that the significant drop in accuracy, from what was expected from the literature review, is due to the smaller sample of data used in the model. Thus, to improve on this research more samples of each label, more comorbidities and patient lab tests will have to be extracted to increase the complexity of the data to fully utilise the GNN modelling. The reason why I believe the issue is in the data extraction phase not the mathematics of the models, is because all the models have better AUC score than a 'random guess'. As we predicting binary outcomes, we would expect a random guess to achieve 50% correct on each of the labels, as all the models have achieved greater than this threshold we can state that the modelling is working and mathematical predictions are correctly taking place, it is just that the data is not large enough for complex patterns to be detected. The non-linearity of the important features also suggests that the issue is the data extraction because linear regression cannot find these non-linear patterns yet, had better results than the more complicated models. Due to time constraints, I was not able to make these changes to the dataset now as we gained access to the dataset late, because of the onboarding process to the NHS. We gained access around a month after the project started losing key weeks of time. As the data server is very large, it would take multiple days, if not weeks to find extra samples of each label, and extra research would have to be taken to find more relevant lab tests that relate to Toxicity. To reach the deadline with a complete project it was not realistic to go back and find more data in the projects time frame.

The late start to receiving the data also hindered the project as in the time frame given, I did not have time to rerun the DNN model with only the top 20 import features. This is a good idea and should be done if further work is given as removing the less important variables can reduce the amount of noise in the model, reducing error. This is beneficial as it will again, increase the AUC score of the model.

Another observation of note is that in every single model, the bottle neck in AUC score was the Toxicity label, as all the models had lower scores on that label than recurrence. This could be because of the lab tests and comorbidities did not relate to that label, thus the modelling found it hard to predict that label. Again, to help with this issue more relevant variables would have to be extracted from the database. When speaking to the Data team at Clatterbridge another potential cause was identified in the Toxicity prediction. When patients come into Clatterbridge for treatment, they are tested for Toxicity once every month, meaning that if a patient's tumour becomes toxic mid-way through the month, and heals in say a week, that toxicity will not be detected and they will be labelled as non-toxic, thus many patients who did experience toxicity will not be recorded as such. As a result, the subset I've deemed as the Toxic subset is almost a random sample of the actual Toxic subset at Clatterbridge, this means that the variables I am looking at will have a degree of randomness to them, adding error into the model and making it harder to predict. We also will not be able to see the full picture of why toxicity occurs, as not all the samples are there, meaning patterns could be harder to detect, in turn again, making prediction by the model much harder.

The computational efficiency of the models is a less important factor in this report, as three of the six models completed to the prediction in less than a minute, being SGC, Logistic Regression and Random Forest, in the table I have inputted the grid search time as that showed more information then placing <1 minute for all. These models can be seen as working instantly and can be placed in a category together. The other 3 models did have relatively different run times, with DNN having the shortest time at 2 minutes and 32 seconds, GCN having 18 minutes and 1 second and GAT having 100 minutes and 38 seconds. Again, this is difficult to compare as the GAT was run on the remote server, this had much more powerful hardware, we can expect that the time taken to run on the laptop, if possible, would have been dramatically longer, more than 10 hours. As we are trying to have the smallest time taken with the best accuracy, we can discard the GCN and GAT models, as they took much longer to run without increasing the accuracy. The 2 most accurate models were Random Forest and DNN, DNN did take a lot longer than Random Forest, so if harsher time constraints were needed, one could take Random Forest as the best overall model as the overall AUC score only dropped by 2% yet ran much quicker. Overall, however all the models ran very quickly, in a realistic setting the run time of the models would not affect their use, thus I would recommend DNN as the best model overall as it had the best AUC score. This is because we were running the models on a powerful PC with GPU integration, if the models were run on worse hardware, one can expect the run times to differ dramatically.

Self-Assessment

Overall, I feel I was able to reach the projects goals successfully, I managed to implement 3 different types of GNN model, the standard model (GCN), a simpler model (SGC) and a complex model (GAT). This gave a strong overview of the different ways the GNN can be implemented, and the different AUC scores were recorded. I then compared those scores to a traditional neural network in the DNN and more basic machine learning models such as Logistic Regression and Random Forests, all implemented correctly. Despite the late start in receiving access to the NHS data, I feel as if time management went well, with all targets being met and the project being completed on time. As previously described, the AUC scores on the models could have been better, however I diagnosed the problem and understand how it could be improved with future work. Again, as this was the first attempt at modelling Clatterbridge's data, never being done before, and I had to construct the data set myself I feel modelling was a success.

As this project was completed in collaboration with Clatterbridge, I communicated with the internal data team through a Microsoft teams group chat. I feel this went well as they were quick to respond to any queries, and issues were quickly resolved. I had to set up meetings to keep the data team up to date with how the project was going, this was new to me but has given me more experience in how to deal with working in a professional environment. The Clatterbridge data was accessed through SQL, this was one of the biggest challenges faced in the project as through my master's course I had learned the very basics of SQL, I had to learn some more advanced techniques of SQL querying, this did slow the data gathering phase of the project as learning a new language can be difficult, but with the help of some example files given by Clatterbridge, I managed to complete this task in the allocated time.

Another challenge I experienced was implementing the graphical neural networks. To implement them, I also had to teach myself the python package PyTorch. I chose this package over TensorFlow because I personally prefer the structure of building the models using classes, the way the code is written makes it easier to read and troubleshoot errors, it also had a secondary package Torch Geometric that made dealing with non-Euclidean graphs more streamlined. This did make programming the models slightly slower however, because I had to take time to learn the correct syntax of the package. Overall, I think this was a good change however as it after I had learned the package tweaking the models and fixing errors became very easy reducing the time taken for those steps.

References

1. **docs.dgl.ai. (n.d.)** https://docs.dgl.ai/en/0.8.x/tutorials/models/1_gnn/9_gat.html
2. **Diaz Ochoa, J.G. and Mustafa, F.E. (2022).**
[doi:https://doi.org/10.1016/j.artmed.2022.102359](https://doi.org/10.1016/j.artmed.2022.102359).
3. **Hastie et al (2001)**
<https://link.springer.com/content/pdf/10.1023/A:1010933404324.pdf>
4. **Katta et al (2023)**
<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10226821/pdf/cureus-0015-00000038301.pdf>
5. **Koehrsen, W. (2020)** <https://williamkoehrsen.medium.com/random-forest-simple-explanation-377895a60d2d>
6. **Lee.org (2024)**
<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9395439>
7. **Lu et al (2022)** <https://dl.acm.org/doi/10.1145/3511616.3513112>
8. **Mayachita, I. (2020)** <https://towardsdatascience.com/understanding-graph-convolutional-networks-for-node-classification-a2bfdb7aba7b>
9. **Saedsayad.com. (2019)** https://www.saedsayad.com/logistic_regression.htm
10. **So Yeon Kim (2023)** <https://www.mdpi.com/2306-5354/10/9/1046>
11. **Veličković et al (n.d.)** <https://arxiv.org/pdf/1710.10903>
12. **Vogt, M et al (2020)** <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7050271/>
13. **Wu et al (n.d.)** <https://arxiv.org/pdf/1902.07153>
14. **www.cancer.gov (2015)** <https://www.cancer.gov/about-cancer/understanding/statistics#:~:text=Approximately%2040.5%25%20of%20men%20and,will%20die%20of%20the%20disease>
15. **Zhang et al (2019)** <https://link.springer.com/article/10.1186/s40649-019-0069-y>
16. **Zhang et al (2019)** <https://link.springer.com/article/10.1186/s40649-019-0069-y>